**Question Bank KEY**

Contents

[1. Objects and classes, Types, The Scanner Class, Using printf and method format 2](#_Toc253341045)

[2. Selections and Conditions. 1](#_Toc253341046)4

[3. Repetitions 1](#_Toc253341047)9

[4. Arrays and Array Lists](#_Toc253341048) 25

[5. Testing and Annotations](#_Toc253341049) 34

[6. UML Diagram](#_Toc253341050) 37

[7. Inheritance](#_Toc253341051) 39

[8. Abstract Classes](#_Toc253341052) 43

[9. Polymorphism](#_Toc253341053) 48

[10. Interfaces](#_Toc253341054) 50

[11. Casting](#_Toc253341055) 54

[12. Exceptions](#_Toc253341056) 56

[13. Sorting](#_Toc253341057) 62

[14. Recursion](#_Toc253341058) 71

[15. Java Collections Framework](#_Toc253341059) 74

[16. Linked Lists](#_Toc253341060) 75

[17. Stacks](#_Toc253341061) 77

[18. Queues](#_Toc253341062) 80

[19. Hashing](#_Toc253341063) 83

[20. Sets and Maps](#_Toc253341064) 87

[21. Trees, Binary Trees, Binary Search Trees](#_Toc253341065) 93

[22. Heaps 11](#_Toc253341066)2

# Objects and classes, Types, The Scanner Class, Using printf and method format

1. We wish to define a class **Wizard**, with the following private instance variables:

**private String wizardName; // name of wizard**

**private int wizardYear; // year the individual became a wizard**

* 1. Complete the code for the constructor shown here. This constructor should assign the values of the parameters **wizardName** and **wizardYear** to the corresponding private instance variables. ***Do not change the supplied code.***

**public Wizard(String wizardName, int wizardYear)**

**{**

**this.wizardName = wizardName;**

**this.wizardYear = wizardYear;**

**}**

* 1. Complete the code for the constructor shown here. This constructor should assign the empty string (**“”**) to **wizardName** and 0 to **wizardYear**. The constructor body should contain a *single* line of code, using the keyword **this** to invoke the constructor you wrote in part a). ***Do not change the supplied code.***

**public Wizard()**

**{**

**this("", 0);**

**}**

1. Suppose we have a class **Book**, with the following private instance variables:

**private String isbn;**

**private String title;**

**private int copyrightDate;**

**private double price;**

Assume we have setters for each of the instance variables and a constructor with four parameters shown here:

**public Book(String isbn, String title, int copyrightDate, double price)**

**{**

**this.setIsbn(isbn);**

**this.setTitle(title);**

**this.setCopyrightDate(copyrightDate);**

**this.setPrice(price);**

**}**

Write a single Java statement to complete the no-arg constructor shown here. Your no-arg constructor should set the isbn to “0-0-0-0”, the title to the empty string, copyright date to 0, and price to 0.0

**public Book()**

**{**

**this("0-0-0-0", "", 0, 0.0);**

**}**

1. Write a single line of code that declares and creates a **Scanner** object named **myReader** that can be used to read user input from the keyboard. Do ***not*** include import statements.

**Scanner myReader = new Scanner(System.in);**

1. Write a code segment to read data from a data file named **strings.txt**, stored in the root folder of the project. Each string in the data file is on a separate line and contains no spaces. As each string is read from the data file, place it in the array list **myStrings** that you created in the previous step. Note: Write only the code segment. No import statements are necessary. Do not include a method header or any “throws” clauses.

**Scanner in = new Scanner(new File("strings.txt"));**

**while(in.hasNext())**

**{**

**myStrings.add(in.next());**

**}**

1. Suppose we have the following data stored in a file named **data.txt**.

**123 5.0**

**hi hello world**

**75 23**

**so long for now**

Suppose also that we have a **Scanner** object named **dataIn** to read from this file.

We then write the following code to read the information from the data file. In the box to the right, use a carat symbol (**^**) to mark the location **dataIn** is pointing to after each of the statements below. ***The first line of code is done for you to illustrate.***

*Note: This is correct, running code. There are no input mismatch problems.*

**dataIn.nextInt(); // carat follows 123 since this line of**

**// code will read the first integer value**

**// of 123 and will then stop**

**dataIn.nextDouble();**

**123^ 5.0^**

**^hi hello world**

**^75 23**

**^so^ long**^ **for now**

**dataIn.nextLine();**

**dataIn.nextLine();**

**dataIn.nextLine();**

**dataIn.next();**

**dataIn.next();**

1. Find the output of the following code segment:

**String str = "saturday is valentine's day";**

**System.out.println(str.indexOf("day"));**

**System.out.println(str.lastIndexOf("day"));**

**System.out.println(str.charAt(2));**

**System.out.println(**

**str.substring(str.indexOf("val")).equals("valentine's day"));**

**System.out.println(str.substring(str.indexOf("t"),**

**str.indexOf("y")).length());**

**System.out.println(str.indexOf("night"));**

**OUTPUT**

**5**

**24**

**t**

**true**

**5**

**-1**

1. Find the output of the following program segment.

**String s1 = "Valentine";**

**OUTPUT**

**true**

**true**

**true**

**false**

**String s2 = "Valentine";**

**String s3 = new String("Valentine");**

**System.out.println(s1.equals(s2));**

**System.out.println(s1.equals(s3));**

**System.out.println(s1 == s2);**

**System.out.println(s1 == s3);**

1. Find the output of the following code segment:

**OUTPUT**

**HARRY POTTER**

**rd of the Ri**

**eathl**

**String r = "Lord of the Rings";**

**String s = " Harry Potter ";**

**String t = "The Deathly Hallows";**

**System.out.println(s.trim().toUpperCase());**

**System.out.println(r.substring(2, r.length() - 3));**

**System.out.println(t.substring(**

**t.indexOf(" ") + 2, t.lastIndexOf(" ") - 1));**

1. Assume we have declared and instantiated the following references to String objects:

**String r = "Northwest Bearcats";**

**String t = "Spring";**

**String u = "Break";**

Evaluate each of the following expressions.

* 1. **r.substring(1,4)=** \_\_\_**ort**\_\_\_\_\_\_\_\_\_\_\_\_\_\_
  2. **u.length()=** \_\_\_\_**5**\_\_\_\_\_\_\_\_\_\_\_\_\_
  3. **r.indexOf("e")=**\_\_\_\_**6**\_\_\_\_\_\_\_\_\_\_\_\_\_
  4. **r.substring( r.indexOf(“west”) ) =** \_\_ **west Bearcats** \_\_
  5. **t.substring( t.indexOf( t.substring( 4 ) ) )** = \_\_\_ **ng** \_\_\_\_
  6. **r.substring( 3, 5 ) + r.substring( 12, 14 )** = ­­­­\_\_\_**thar** \_\_\_\_

1. Assume we have declared and instantiated the following variables:

**String str1 = new String ("Hyderabad, India");**

**String str2 = new String("Leonard Cohen");**

**int a = 5;**

**int b = 22;**

**int c = 4;**

Evaluate each of the following expressions:

* 1. **str2.lastIndexOf('o')**

**Answer: \_\_\_\_\_\_\_\_\_9\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **str1.substring(6,9) + str2.substring(10)**

**Answer: \_\_\_\_\_\_\_\_\_badhen\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **str1.substring(str1.indexOf(','))**

**Answer: \_\_\_\_\_\_\_\_\_, India\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **a + b % c \* b / a**

**Answer: \_\_\_\_\_\_\_\_\_13\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **a < c && b < c \* a || !(a + b < c)**

**Answer: \_\_\_\_\_\_\_\_\_true\_\_\_\_\_\_\_\_\_\_\_\_**

1. Consider the class definition shown here:

**public class BankAccount**

**{**

**private String accountID;**

**private String customerName;**

**private double accountBalance;**

**public BankAccount(String accountID, String customerName,**

**double accountBalance)**

**{**

**this.accountID = accountID;**

**this.customerName = customerName;**

**this.accountBalance = accountBalance;**

**}**

**public void deposit(double amount)**

**{**

**setAccountBalance(accountBalance + amount);**

**}**

**}**

* 1. Write the prototype for method **deposit**:

**Answer: \_\_\_public void deposit(double amount)\_\_\_\_**

* 1. Write the signature for method **deposit**:

**Answer: \_\_\_deposit(double)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. In the space below, write the code for the getter method for accountID. You must include the method header.

**public String getAccountID()**

**{**

**return accountID;**

**}**

* 1. In the space below, write the code for a no-arg constructor. This constructor will set private instance variables of type **String** to the empty string. Private instance variables of type **double** will be set to 0.0. Write the code for the no-arg constructor in the space below. You must include the header line for the constructor.

**public BankAccount()**

**{**

**this("", "", 0.0);**

**}**

*Or the body can be replaced with*

**accountID = "";**

**customerName = "";**

**accountBalance = 0.0;**

1. Consider the following method:

**public void setIsbn(String isbn)**

**{**

**this.isbn = isbn;**

**}**

* 1. In the space below, write the prototype for this method.

**public void setIsbn(String isbn)**

* 1. In the space below, write the signature for this method.

**setIsbn(String)**

1. Write the prototype of the **toString** method from the problem above.

**public String toString()**

1. Assume we have declared and instantiated the following references to String objects.

**String a = "Go ahead, make my day";**

**String b = "Here's looking at you kid";**

**String c = " WOW!!!!! "; // there are four spaces**

**// at beginning and end**

Evaluate each of the following expressions:

* 1. **c.trim().substring(3, 6) !!!**
  2. **b.substring(b.lastIndexOf("o")) ou kid**
  3. **a.substring(0, a.indexOf(",")).concat(**

**b.substring(b.lastIndexOf(" ")))**

**Go ahead kid**

1. Assume **str** has been declared and instantiated as the name of a person. The name consists of at least two, but possibly more words, with pairs of words separated by a single space. There are no leading or trailing spaces. Complete the *single* line of code below that stores the first and last initials in a **String** variable named **initials**. A period should follow each initial and the two initials should be separated by a single space.

Here are some examples of values of **str** and the resulting value that should be stored in **initials**.

|  |  |
| --- | --- |
| **Str** | **initials** |
| **Katherine Hepburn** | **K. H.** |
| **Humphrey DeForest Bogart** | **H. B.** |
| **Maggie Elizabeth Suzanne Charlene Thomas** | **M. T.** |

Complete this code. You can only have a single line of code. Use only **String** methods. Do not use loops or if statements.

**String initials = str.substring(0, 1) + ". " +**

**str.substring(str.lastIndexOf(" ") + 1,**

**str.lastIndexOf(" ") + 2) + ".";**

1. Write a single line of code that creates a **Scanner** object named **myReader** that can be used to read from the keyboard.

**Scanner myReader = new Scanner(System.in);**

1. Consider the following code for the **Dog** class:

**public class Dog**

**{**

**private String name;**

**private int age;**

**public Dog()**

**{ ... }**

**public Dog(String name, int age)**

**{ ... }**

**public void incrementAge(int amount)**

**{ ... }**

**public void decrementAge(int amount)**

**{ ... }**

**public String dogDescription()**

**{ ... }**

**}**

* 1. How many methods does this class have? Do not include constructors in your count. 3
  2. What is the type of the second attribute of the constructor with parameters? int
  3. Write the code for a setter method for the attribute **age**. Include the method header.

**public void setAge(int age)**

**{**

**this.age = age;**

**}**

1. Find the output of the following code segment.

**0**

**6**

**900**

**int a = 30;**

**int b = 4;**

**int c = 12;**

**System.out.println(b / c);**

**System.out.println(a / c + b);**

**System.out.println(a % b + a / c + a \* a - b);**

1. Evaluate each of the arithmetic expressions below.
   1. **40 - 3 - 5 \* 7 = \_\_\_\_2\_\_\_\_\_**
   2. **13 / 4 + 3 \* 4 % 5 = \_\_\_\_5\_\_\_\_\_**
   3. **2.0 / 4.0 + 3.0 \* 2.0 = \_\_\_\_6.5\_\_\_\_\_**
2. Suppose we have a Java class named **Dog**. In order to compile this program using a command prompt window, use the command
   1. **compile Dog.java**
   2. **java Dog.java**
   3. **javac Dog.java**
   4. **class Dog.java**
3. The \_\_\_\_\_\_\_\_\_ translates the Java bytecode in the class file into machine language and runs the program.
   1. Java Virtual Machine
   2. Java Compiler
   3. Class Compiler
   4. Javac Machine
4. Which of the following is/are true of the **println** method in Java?
   1. it prints the value specified in the argument and adds a line separator
   2. it is overloaded; that is, there are several different versions with different types of parameters
   3. both of the above are true
   4. none of the above are true
5. Suppose we have created a **GregorainCalendar** object containing today’s date, using the following statement:

**GregorianCalendar cal = new GregorianCalendar();**

Write a single Java statement that will use this object to find the current year and will store this value in an **int** variable named **current**. Your code must work, regardless of when this code is run. If it is run this year, the value 2011 will be stored in **current**. If we run it next year, the value 2012 will be stored in **current**.

**int current = cal.get(GregorianCalendar.YEAR);**

1. Suppose we have a class **Book**, with the following private instance variables, along with getters for each one:

**private String isbn;**

**private String title;**

**private int copyrightDate;**

**private double price;**

Write the code for the **toString** method. Include the method header and the appropriate annotation. This method returns a formatted string, with isbn right justified in a field of width 15, title left justified in a field of width 50, copyright date right justified in a field of width 4, and price right justified in a field of width 5, with 2 decimal places and preceded by a dollar sign ($).

**@Override**

**public String toString()**

**{**

**return String.format("%15s %-50s %4d $%5.2f", this.getIsbn(),**

**this.getTitle(), this.getCopyrightDate(), this.getPrice());**

**}**

1. In Java, numbers, such as **int**s, **double**s, and **float**s, are examples of
   1. reference variables
   2. classes
   3. methods
   4. primitive types
2. The signature of the **indexOf** method of the **String** class, used to return the index of the first occurrence of the specified substring is
   1. **public int indexOf (String str)**
   2. **int indexOf (String str)**
   3. **indexOf (String)**
   4. **indexOf()**
3. Which of the following is/are true of constructors?
   1. when you define a class, if you include a constructor with parameters, you must also include a no-arg constructor
   2. if you define a class without a constructor, Java provides a no-arg constructor
   3. both of the above are true
   4. none of the above are true
4. To declare a constant in Java, use the keyword
   1. var
   2. constant
   3. const
   4. final
5. Which of the following is/are true about **int**s in Java?
   1. **int** is a class
   2. **int** is a primitive type
   3. both a and b are true
   4. none of the above are true
6. \_\_\_\_ provide a way to store the state of an object.
   1. Attributes
   2. Classes
   3. Constructors
   4. Methods
7. The signature of a method contains
   1. the name of the method only
   2. the name of the method and the list of parameter data types
   3. the return type, the name of the method and the list of parameters
   4. the access specifier, the return type, the name of the method and the list of parameters
8. A constructor with no parameters is called a(n)
   1. default constructor
   2. empty constructor
   3. no-arg constructor
   4. simple constructor
9. To create an object in Java, you normally use the \_\_ operator.
   1. begin
   2. construct
   3. create
   4. new
10. In a class definition, the implicit parameter used with a method or instance variable can be accessed using the keyword
    1. param
    2. private
    3. this
    4. void
11. Character literals are denoted by double quote marks.
    1. true
    2. false
12. Suppose we have declared and initialized **number** as shown here:

**int number = 17;**

Which of the following statements can be used to cast to a double and print? (Output will be **17.0**.)

* 1. **System.out.println(number);**
  2. **System.out.println((double) number);**
  3. **System.out.println(double number);**
  4. **System.out.println(number(double));**

1. To determine if two strings represent the same sequence of characters, use
   1. **==**
   2. **method equals**
   3. **method toString**
   4. **method stringEquals**
2. In Java, the compiler translates the source program written in Java into an intermediate language called
   1. Java virtual code
   2. Java bytecode
   3. Java assembler code
   4. Java source code
3. To compile a Java program in a command prompt window, use the command
   1. compile
   2. java
   3. java assemble
   4. javac
4. Java classes provide \_\_\_ that allow you to create new objects.
   1. attributes
   2. entities
   3. constructors
   4. methods
5. A class may have multiple constructors.
   1. true
   2. false

# Selections and Conditions

1. Evaluate each of the following expressions. You ***must*** show ***all*** work and you ***must*** circle or underline your answer for full credit.
   1. **50 + 20 / 3 – 17 % 5 \* 2**

**50 + 6 - 2 \* 2 = 50 + 6 - 4 = 56 - 4 = 52**

* 1. **10 > 2 && 18 / 4 < 2 || !(5 < 6 || 3 > 10 % 2 && !(4 > 1))**

**10 > 2 && 4 < 2 || !(5 < 6 || 3 > 0 && !(4 > 1))**

**T && F || !(T ||T && !T)**

**T && F || !(T || T && F)**

**T && F || !(T || F)**

**T && F || !(T)**

**T && F || F**

**F || F**

**F**

1. Evaluate the following expression, assuming the declarations shown here:

**int a = 3;**

**int b = 4;**

**int c = 23;**

**(c % b) + (a + b) / 2 \* a - b - c = \_\_-15\_\_\_\_\_\_\_\_\_\_\_\_**

**(23 % 4) + (3 + 4) / 2 \* 3 – 4 – 23**

**3 + 7 / 2 \* 3 – 4 - 23**

**3 + 3 \* 3 – 4 – 23**

**3 + 9 – 4 – 23**

**12 – 4 – 23**

1. **– 23**

**-15**

1. Find the output of the following code segment:

**OUTPUT**

**false**

**int a = 7;**

**int b = 3;**

**int c =4;**

**System.out.println(**

**a + b \* c % 2 < a - b - c / 3 && a > b || c - b > a / 2);**

1. Find the output of the following code segment:

**int a = 15;**

**int b = 10;**

**OUTPUT**

**true**

**int c = 4;**

**System.out.println(**

**(a + b) % c > b - a / 2 && b > 2 \* c || !(b > a));**

**(a + b) % c > b - a / 2 && b > 2 \* a || !(b > a)**

**(15 + 10) % 4 > 10 – 15 / 2 && 10 > 2 \* 15 || !(10 > 15)**

**25 % 4 > 10 – 15 / 2 && 10 > 2 \* 15 || !(10 > 15)**

**1 > 10 – 7 && 10 > 30 || !(10 > 15)**

**1 > 3 && 10 > 30 || !(10 > 15)**

**F && F || !F**

**F && F || T**

**F || T**

**T**

1. Find the output of the following program segment, assuming num is initialized to the values given in each of the OUTPUT boxes.

**Output if num is initialized to 43**

**8**

**int num = \_\_\_\_;**

**if(num > 100)**

**{**

**num = num / 2;**

**Output if num is initialized to 25**

**27**

**} else if (num > 50)**

**{**

**num = num /3 ;**

**} else if(num > 25)**

**Output if num is initialized to 101**

**50**

**{**

**num = num / 5;**

**} else**

**{**

**Output if num is initialized to 51**

**17**

**num += 2;**

**}**

**System.out.println(num);**

1. Find the output of this program segment.

**int a = 10;**

**OUTPUT**

**12**

**int b = 17;**

**System.out.println(**

**a > b ? a + 3: b - 5);**

1. Find the output of the following code segment.

**OUTPUT**

**Not less**

**int number = 5;**

**if(number < 5)**

**{**

**System.out.println("Less");**

**} else**

**{**

**System.out.println("Not less");**

**}**

1. Assume the variable **num** has been declared to be of type **int**. The following Java code segment contains a nested if statement. Use the code segment to answer the questions following the code.

**if (num > 500)**

**{**

**num = num + 2;**

**} else if (num > 100)**

**{**

**num = num - 5;**

**} else**

**{**

**num = num \* 2;**

**}**

**System.out.println("num = " + num);**

Determine the output assuming **num** contains each of the following values:

* + - * 1. (b) (c)

**num = 100; num = 1000; num = 300;**

**OUTPUT**

**num = 295**

**OUTPUT**

**num = 1002**

**OUTPUT**

**num = 200**

1. Consider the following code segment.

**if(num < 10)**

**{**

**num = num + 5;**

**if(num > 10)**

**{**

**num = num / 2;**

**} else**

**{**

**num = num \* 2;**

**}**

**} else**

**{**

**num = num \* 3;**

**if(num < 50)**

**{**

**num = num - 7;**

**} else**

**{**

**num = num / 7;**

**}**

**} // end if**

**System.out.println(num);**

* 1. Find the output if **num** is initialized to 20. 8
  2. Find the output if **num** is initialized to 3. 16

1. In evaluating conditional expressions, which of the following has the highest precedence?
   1. relational operators
   2. logical operators
   3. arithmetic operators
2. What is the output of the following code segment?

**int number = 3;**

**System.out.println(number == 3 ? "four" : "three");**

* 1. **three**
  2. **four**
  3. **four : three**
  4. there is no output

1. Assume there is an **int** variable named **creditHours** with a value stored in it. Write a single Java statement using the conditional operator (**? :**) that prints the value stored in **creditHours**, followed by a space, followed by **credit hour** if the value stored in **creditHours** is 1, and **credit hours** otherwise.

**System.out.println(creditHours + " " +**

**(creditHours == 1 ? "credit hour" : "credit hours"));**

1. Explain the conditional operator (**? :**) in Java.
   1. First, describe the conditional operator in words. What is the syntax of this operator and what happens when the statement is executed?

Syntax: **condition ? return value if true : return value if false**

A condition precedes the **?**. The condition is tested to yield a boolean value of **true** or **false**. If the value returned by the condition is true, then the value preceding the colon (**:**) is returned by this statement; otherwise, the value following the colon is returned.

* 1. Second, give an example of a Java code segment, using the conditional operator. Do not write an entire method. Simply write a short segment, of two or three statements at most, showing how to use the conditional operator.

**int hoursWorked = 10;**

**System.out.println("Hours worked = " + hoursWorked +**

**(hoursWorked == 1 ? " hour" : " hours"));**

# Repetitions

1. Find the output of the following code segment:

**Output**

**0 33**

**-5 38**

**-10 43**

**int num1 = 25;**

**int num2 = 14;**

**for(int count = 0; count < 3; count++)**

**{**

**while(num1 > 10)**

**{**

**num1 -= 10;**

**while(num2 < 50)**

**{**

**num2 += 10;**

**}**

**num2 /= 2;**

**}**

**num1 -= 5;**

**num2 += 5;**

**System.out.println(num1 + " " + num2);**

**}**

1. Find the output of this program segment.

**OUTPUT**

**outer: 4**

**inner: 20**

**inner: 25**

**outer: 8**

**inner: 29**

**outer: 16**

**outer: 32**

**out of loop and inner = 31**

**out of loop and outer = 64**

**int inner = 20;**

**int outer = 4;**

**while(outer < 50)**

**{**

**System.out.println(**

**"outer: " + outer);**

**outer = outer \* 2;**

**while(inner < 30)**

**{**

**System.out.println(**

**"inner: " + inner);**

**inner = inner + 5;**

**}**

**inner = inner - 1;**

**}**

**System.out.println(**

**"out of loop and inner = " + inner);**

**System.out.println(**

**"out of loop and outer = " + outer);**

1. Find the output of this program segment.

**OUTPUT**

**i = 0**

**m = 2**

**m = 1**

**i = 5**

**m = 2**

**m = 1**

**for(int i = 0; i < 10; i = i + 5)**

**{**

**System.out.println("i = " + i);**

**for(int m = 2; m > 0; m = m - 1)**

**{**

**System.out.println("m = " + m);**

**}**

**}**

1. Find the output of the following code segment.

**int myVar = 1;**

**int count = 4;**

**OUTPUT**

**24 16**

**while(count < 20)**

**{**

**count += 5;**

**myVar \*= 2;**

**}**

**System.out.println(count + " " + myVar);**

1. In this problem, you will be asked to trace the code by showing the value of each variable. Here is an example of a tracing problem

**int a = 5;**

**a b i**

**5 20 0**

**4 19 1**

**3 18 2**

**2 3**

**1 4**

**int b = 20;**

**for(int i = 0; i < 4; i++)**

**{**

**a--;**

**if (i % 2 == 0)**

**{**

**b--;**

**}**

**System.out.println(a + " " + b);**

**}**

The trace to the right shows all the values that each of the three variables are assigned as the program runs. Note that the output is not shown.

Your task is to trace the program shown below. Place your answer in the location provided.

**int num1 = 10;**

**num1 num2 sum i**

**10 14 0 0**

**7 11 21 1**

**8 42 2**

**4 60 0**

**78 1**

**89 2**

**100 0**

**1**

**2**

**int num2 = 14;**

**int sum = 0;**

**while(num1 + num2 > 15)**

**{**

**if(num1 < num2)**

**{**

**num2 -= 3;**

**} else**

**{**

**num2 -= 4;**

**num1 -= 3;**

**}**

**for(int i = 0; i < 2; i++)**

**{**

**sum += num1 + num2;**

**}**

**}**

1. Find the output of the following code segment.

**OUTPUT**

**myVar = 10**

**int myVar = 0;**

**for(int count = 0; count < 5; count++)**

**{**

**myVar += count;**

**}**

**System.out.println("myVar = " + myVar);**

1. Find the output of the following code segment.

**OUTPUT**

**outer = 20**

**inner = 1**

**inner = 3**

**outer = 13**

**inner = 1**

**inner = 3**

**outer = 6**

**inner = 1**

**inner = 3**

**1 -1**

**int outer = 20;**

**int inner = 1;**

**while(outer > 0)**

**{**

**System.out.println("outer = " + outer);**

**while(inner < 5)**

**{**

**System.out.println("inner = " + inner);**

**inner += 2;**

**}**

**inner = 1;**

**outer -= 7;**

**}**

**System.out.println(inner + " " + outer**

**OUTPUT**

**5 0 1 0**

**12 3 8 14**

**15 6 8 14**

**18 12 8 14**

**23 24 9 14**

1. Find the output of the following code segment.

**int m = 0;**

**int n = 0;**

**int s = 0;**

**int w = 0;**

**while(m < 20)**

**{**

**if(m == 5)**

**{**

**for(int i = 0; i < 3; i++)**

**{**

**m = m + s;**

**n += 1;**

**s \*= 2;**

**w = w + s;**

**}**

**System.out.println(m + " " + n + " " + s + " " + w);**

**} else if(n > 0 && n < 10)**

**{**

**m += 3;**

**n \*= 2;**

**System.out.println(m + " " + n + " " + s + " " + w);**

**} else**

**{**

**m += 5;**

**n \*= 2;**

**s += 1;**

**System.out.println(m + " " + n + " " + s + " " + w);**

**}**

**}**

1. Find the output of the following code segment:

**5 6 7 8 9**

**for(int count = 5; count < 10; count++)**

**{**

**int sum = 0;**

**sum += count;**

**System.out.print(count + " ");**

**}**

1. Find the output of the following code segment if **counter** is initialized to 11 and **myNum** is initialized

to 8.

**while(counter < 50)**

**counter: 21 myNum: 9**

**counter: 28 myNum: 8**

**counter: 34 myNum: 7**

**counter: 39 myNum: 6**

**counter: 43 myNum: 6**

**counter: 47 myNum: 6**

**counter: 51 myNum: 6**

**{**

**counter += myNum;**

**while(myNum < 7)**

**{**

**myNum++;**

**}**

**if(myNum < 10 && counter < 25)**

**{**

**myNum += 1;**

**counter += 2;**

**} else**

**{**

**myNum -= 1;**

**counter -= 2;**

**}**

**System.out.print("counter: " + counter + " ");**

**System.out.println("myNum: " + myNum);**

**}**

1. Find the output of the following code segment:

**Output**

**9 4**

**6 10**

**9 8**

**6 15**

**12 11**

**15 4**

**int num1 = 0;**

**int num2 = 10;**

**for(int i = 0; i < 3; i++)**

**{**

**while (num1 <= num2)**

**{**

**num1 += 3;**

**num2 -= 2;**

**}**

**System.out.println(num1 + " " + num2);**

**num1 \*= 2;**

**if(num1 > 20)**

**{**

**num1 += 4 + i;**

**num2 = num2 - 5 - i;**

**} else if(num1 > 10)**

**{**

**num1 = num1 - 5 - i;**

**num2 += 6 + i;**

**} else**

**{**

**num1 += 10;**

**num2 -= 7;**

**}**

**num1 /= 2;**

**System.out.println(num1 + " " + num2)**

1. Find the output of the following code segment:

**Output**

**i = 0**

**i = 1**

**start = 24**

**i = 0**

**i = 1**

**start = 37**

**i = 0**

**i = 1**

**start = 56**

**i = 0**

**i = 1**

**start = 69**

**i = 0**

**i = 1**

**start = 88**

**i = 0**

**i = 1**

**start = 101**

**int start = 5;**

**while(start < 100)**

**{**

**for(int i = 0; i < 2; i++)**

**{**

**System.out.println("i = " + i);**

**}**

**start = start + 17;**

**if(start % 2 == 0)**

**{**

**start = start + 2;**

**} else**

**{**

**start -= 4;**

**}**

**System.out.println("start = " + start);**

**}**

# Arrays and Array Lists

1. This problem asks you to write statements using arrays in Java.
   1. Write a single Java statement to declare and initialize an array of **int** values. The array should be named **myArray** and should hold 15 values.

**int[] myArray = new int[15];**

* 1. Write a traditional **for** loop (that is, do ***not*** use an enhanced **for** loop) that will fill the array you created in part a). The value stored in each entry should be the result of multiplying the index by 3. That is, the value stored at index 0 will be 3 \* 0, which is 0; the value stored at index 1 will be 3 \* 1, which is 3; and so forth.  ***Your solution must work even if the length of the array is changed.***

**for(int i = 0; i < myArray.length; i++)**

**{**

**myArray[i] = 3 \* i;**

**}**

1. Write a single Java statement to create an array list of **String** values named **myStrings**.

**ArrayList<String> myStrings = new ArrayList<String>();**

1. Write a single Java statement to create an array of **int** values named **intList**. The array should hold 50 values.

**int[] intList = new int[50];**

1. Write an enhanced for loop to print the contents of the array list **myStrings** that you created and filled in the previous problems. Print each string on a separate line.

**for(String str : myStrings)**

**{**

**System.out.println(str);**

**}**

1. Write an “ordinary” (that is, *not* enhanced) for loop to print the contents of the array list **myStrings** that you created and filled in the previous problems. Print each string on a separate line.

**for(int count = 0; count < myStrings.size(); count++)**

**{**

**System.out.println(myStrings.get(count));**

**}**

1. Assume we have class **Project** as shown here:

**public class Project**

**{**

**private String projectId;**

**private String projectName;**

**private int hoursRequired;**

**public Project(String projectId, String projectName,**

**int hoursRequired)**

**{**

**this.projectId = projectId;**

**this.projectName = projectName;**

**this.hoursRequired = hoursRequired;**

**}**

**public int getHoursRequired()**

**{**

**return hoursRequired;**

**}**

**@Override**

**public String toString()**

**{**

**return projectId + " " + projectName + " " + hoursRequired;**

**}**

**}**

Assume we have written a driver program. In the driver program, we have created an array list of projects named **myProjects**, and we have added several projects to **myProjects**. (This is already done – do NOT write the code to do this.)

Write a code segment that finds and prints the total of the hours required for all projects in the array list. For example, if we have two projects in the list, and one requires 50 hours and the other requires 140 hours, then your code segment will print 190. Your code segment must work for any number of projects that may be stored in the array list.

**int sum = 0;**

**for(Project proj : myProjects)**

**{**

**sum += proj.getHoursRequired();**

**}**

**System.out.println(sum);**

1. Suppose we have a class **Horse**, with the following private instance variables:

**private String regNum;**

**private char checkDigit;**

**private String name;**

**private double price;**

Complete the code for the constructor and methods for this class described below.

**/\*\***

**\* Constructor. Sets the values of the private instance variables**

**\* regNum, name, and price.**

**\* @param regNum The value used to set regNum.**

**\* @param name The value used to set name.**

**\* @param price The value used to set price.**

**\*/**

**public Horse(String regNum, String name, double price)**

**{**

**this.regNum = regNum;**

**this.name = name;**

**this.price = price;**

**}**

**/\*\***

**\* Sets the value of the private instance variable checkDigit.**

**\* The check digit is the sum of the digits of the registration**

**\* number (regNum), modulo 10.**

**\* Note: Must use a loop; cannot store digits in separate variables.**

**\* Examples:**

**\* 11123450: sum of digits is 17; check digit is 7 (17 % 10)**

**\* 101010: sum of digits is 3; check digit is 3 (3 % 10)**

**\* 11345060: sum of digits is 20; check digit is 0 (20 % 10)**

**\*/**

**public void setCheckDigit()**

**{**

**int sum = 0;**

**for(int i = 0; i < regNum.length(); i++)**

**{**

**sum += (regNum.charAt(i) - 48);**

**}**

**checkDigit = (char)((sum % 10) + 48);**

**}**

**/\*\***

**\* Returns a String consisting of the full registration number.**

**\* The full registration number is the value stored in regNum,**

**\* followed by a hyphen, followed by the check digit.**

**\* Examples:**

**\* If regNum is 11123450, fullRegNum returns 11123450-7**

**\* If regNum is 101010, fullRegNum returns 101010-3**

**\* If regNum is 11345060, fullRegNum returns 11345060-0**

**\* @return a String consisting of the full registration number.**

**\*/**

**public String fullRegNum()**

**{**

**return regNum + "-" + checkDigit;**

**}**

**/\*\***

**\* Returns a String representation of this Horse.**

**\* The String representation consists of the name of the horse,**

**\* left justified in a field of width approximately 25, followed**

**\* by the full registration number (this is the value returned**

**\* by fullRegNum), left justified in a field of width**

**\* approximately 12, followed by the price of the horse,**

**\* right justified in a field of width 9, with 2 digits**

**\* following the decimal point. The price should be preceded**

**\* by a dollar sign ($).**

**\* @return a String representation of this Horse**

**\*/**

**@Override**

**public String toString()**

**{**

**return String.format("%-25s %-12s $%9.2f", name, fullRegNum(),**

**price);**

**}**

1. Fill in method **main** as described below. This method fills an array list with integers read from a data file and prints the values. You may assume that all necessary import statements have been included.

**public static void main(String[] args) throws FileNotFoundException**

**{**

**// Create an ArrayList of Integers named myInts**

**ArrayList<Integer> myInts = new ArrayList<Integer>();**

**// Create a Scanner object that can be used to read**

**// from a data file named "numbers.txt". The file**

**// will be placed in the root of the project folder.**

**Scanner in = new Scanner(new File("numbers.txt"));**

**// Write a loop to read from the data file. As each**

**// integer value is read, it is placed in the array list**

**// myInts.**

**// The data file consists of a list of integer values, each**

**// on a separate line. Your code must work regardless of**

**// how many integers are in the list.**

**while(in.hasNext())**

**{**

**myInts.add(in.nextInt());**

**}**

**// Write an enhanced for loop to print the values stored in**

**// the array list. Each value should be printed on a separate**

**// line.**

**for(int num : myInts)**

**{**

**System.out.println(num);**

**}**

**// Write a loop to print the values stored in the array list**

**// in reverse order. All integers should be printed on the same**

**// line with a space between pairs of integers.**

**for(int i = myInts.size() - 1; i >= 0; i--)**

**{**

**System.out.print(myInts.get(i) + " ");**

**}**

**} // end method main**

1. Fill in method **main** as described below. This method fills an array with doubles. You may assume that all necessary import statements have been included.

**public static void main(String[] args)**

**{**

**// Create an array of double values of length 10 named**

**// myDoubles.**

**double[] myDoubles = new double[10];**

**// Write a for loop that will fill the array myDoubles. The**

**// value for each entry in the array must be one-half of the**

**// index. So the value in location 0 will be 0.0; the value**

**// in location 1 will be 0.5; the value in location 2 will**

**// be 1.0; the value in location 3 will be 1.5, and so forth.**

**// Your code must work even if we change the length of the**

**// array.**

**for(int i = 0; i < myDoubles.length; i++)**

**{**

**myDoubles[i] = i / 2.0; // or i \* 0.5**

**}**

**// Write an enhanced for loop that prints the values stored**

**// in the array, with each value on a separate line.**

**for(double num : myDoubles)**

**{**

**System.out.println(num);**

**}**

**} // end method main**

1. This problem asks you to write statements using ArrayLists in Java.
   1. Write a single Java statement to declare and initialize an **ArrayList** of **String** values. The array should be named **myStrings**. Do not specify the capacity. Accept the default capacity for the size of the arraylist.

**ArrayList<String> myStrings = new ArrayList<String>();**

* 1. Write three Java statements to add elements to the arraylist **myStrings** that you created in part a). Your statements should result in the string **midge** being placed in the array list at index 0; the string **carmen** at index 1; the string **leonard** at index 2.

**myStrings.add("midge");**

**myStrings.add("carmen");**

**myStrings.add("leonard");**

* 1. Write an *enhanced* **for** loop that will print the values in the arraylist **myStrings**. Print each value on a separate line.

**for(String s : myStrings)**

**{**

**System.out.println(s);**

**}**

* 1. Write a traditional **for** loop (that is, do *not* use an enhanced **for** loop) that will print the values in the arraylist **myStrings**. All values should be printed on the same line, with pairs of values separated by a single space.

**for(int i = 0; i < myStrings.size(); i++)**

**{**

**System.out.print(myStrings.get(i) + " ");**

**}**

1. This problem asks you to write statements using two-dimensional arrays in Java.
   1. Write a single Java statement to declare and initialize a two-dimensional array of **int** values. The array should be named **myInts**. It should have 3 rows and 7 columns.

**int[][] myInts = new int[3][7];**

* 1. Assume the the two-dimensional array that you created in the previous step has been filled with values. Do ***not*** write the code to do this. Your task is to write code to print the values stored in the two-dimensional array.

When printing the values in the array, the values stored in a row should be printed on a single line, with pairs of numbers separated by a single space. Each row will be on a separate line. ***Your code must work for any number of rows and columns.***

**for(int row = 0; row < myInts.length; row++)**

**{**

**for(int col = 0; col < myInts[row].length; col++)**

**{**

**System.out.print(myInts[row][col] + " ");**

**}**

**System.out.println();**

**}**

1. Assume that the array **intList** that you created in a previous problem has been filled with values. Write a for loop to print the contents of all entries in the array that are indexed by an even number (0, 2, 4, …). All values should be printed on the same line, separated by a single space. Your code should work even if we declare the array to hold a different number of elements.

**for(int count = 0; count < intList.length; count += 2)**

**{**

**System.out.print(intList[count] + " ");**

**}**

* 1. In the space below write a *single* Java statement to declare and create an array of **String** values. The array should be able to store 15 strings. The name of the array should be **myStrings**.

**String[] myStrings = new String[15];**

* 1. For the array **myStrings** that you created in the previous step, write a *single* Java statement that will store the string **Leonard** in the position indexed by 5.

**myStrings[5] = "Leonard";**

* 1. In the space below write a *single* Java statement to declare and create an array list of **String** values. The name of the array list should be **myList**.

**ArrayList<String> myList = new ArrayList<String>();**

* 1. Assume that several strings have been added to **myList** (created in step a), so it now looks similar to this. However, there may be more or fewer strings than shown here. That is, do not assume that you know that exactly five strings have been added.

**myList**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **Zelda** | **Midge** | **Eve** | **Carmen** | **Leonard** |  | **…** |

Write a *single* Java statement that will store the string **Fudge** at the end of the current array list. In the example above **Fudge** would appear in location 5, but remember that the list may contain more or fewer items than shown here. Your statement must store the string **Fudge** at the end of the current list, regardless of the current size.

**myList.add("Fudge");**

* 1. Assume that a minimum of 5 items have been added to **myList**, created in step a. Write a *single* Java statement to replace the string currently at index 1 with the string **Audrey**.

**myList.set(1, "Audrey");**

* 1. Write a traditional **for** loop (that is, do not use an enhanced **for** loop) that will print out all entries in the array list **myList**, created in step a. The entries should all be on a single line, with one space separating each pair of strings. Do not make any assumptions about the size of the array list. Your code should work regardless of the number of strings currently stored in the list.

**for(int i = 0; i < myList.size(); i++)**

**{**

**System.out.print(myList.get(i) + " ");**

**}**

* 1. Same as the previous problem, but this time use an enhanced **for** loop.

**for(String str : myList)**

**{**

**System.out.print(str + " ");**

**}**

1. Assume we have an array list of strings declared and instantiated as follows:

**ArrayList<String> myStrings = new ArrayList<String>();**

* 1. Also assume we have a **Scanner** object named **inString** that reads from a file of strings. (Do not write the statement to create the **Scanner** object. We are assuming it has already been created.) You may assume that each string is on a separate line and contains no white space.

Write a code segment to read the strings in the data file. As each string is read, add it to the end of the array list **myStrings**.

**while(inString.hasNext())**

**{**

**myStrings.add(inString.next());**

**}**

* 1. Write the code for an enhanced for **loop** that will read the strings stored in **myStrings** and will print them on a single line, with pairs of strings separated by a single space.

**for(String str : myStrings)**

**{**

**System.out.print(str + " ");**

**}**

* 1. Write a single Java statement that will insert the string **Sherman** into **myStrings** at the location indexed by 2. Note that this is an insert operation, not a replace.

**myStrings.add(2, "Sherman");**

* 1. Write a single Java statement that will return the value currently stored at the location indexed by 4 in **myStrings**.

**myStrings.get(4);**

1. Suppose we have declared and instantiated an array as follows:

**int[] myNumbers = new int[6];**

In addition, assume we have filled the array with values. Write a single Java statement that will print the value that is stored in the array at the location indexed by 2.

**System.out.println(myNumbers[2]);**

1. An array in Java has a fixed length.
   1. true
   2. false
2. In Java, an array is not an object.
   1. true
   2. false
3. Which of the following is true about array lists?
   1. you cannot leave empty spaces in an array list
   2. array lists have a fixed size
   3. both a and b are true
   4. none of the above are true

# Testing and Annotations

1. Suppose we have a program in which we input the name of a city and its current tax rate and then take certain actions based on the input value. The tax rate input should be greater than or equal to 0% and less than or equal to 100%. Depending on the tax rate, we output a message as shown here:

|  |  |
| --- | --- |
| Value Input as the Tax Rate | Output Message |
| < 0 OR > 100 | ILLEGAL TAX RATE |
| >= 0 AND < 5 | VERY LOW TAX RATE |
| >= 5 AND < 10 | LOW TAX RATE |
| >= 10 AND < 15 | AVERAGE TAX RATE |
| >= 15 AND < 25 | HIGH TAX RATE |
| >= 25 AND < 50 | VERY HIGH TAX RATE |
| >= 50 AND <= 100 | EXTREMELY HIGH TAX RATE |

In order to test the program, we must, among other things, check all boundary values to ensure that the code is correct for all such values. In the space below, list the boundary values to be checked. Do *not* write the code for this problem.

Answer (list of boundary values):0, 5, 10, 15, 25, 50, 100

1. The method shown below allows the user to send a department number as the argument. It then looks through some course enrollment information and returns the total enrollment for courses in the specified department. In the space below, write the Javadoc comments that should precede this method.

**/\*\***

**\* Finds the total enrollment for a specified department.**

**\* @param deptNum The department number for which we are returning**

**\* the total enrollment.**

**\* @return an int value representing the total enrollment for a**

**\* specified department.**

**\*/**

**public int enrollmentByDept(String deptNum)**

**{**

**int enrollment = 0;**

**for (Course crs : courseListing)**

**{**

**if (crs.getDeptNumber().equals(deptNum))**

**{**

**enrollment += crs.getEnrolled();**

**}**

**}**

**return enrollment;**

**}**

1. The method shown below multiplies two integer values and returns the result. In the space provided on your answer sheet, write the Javadoc comments that would precede this method.

**/\*\***

**\* Returns the product of the two numbers passed as parameters.**

**\* @param num1 The first of the two numbers to multiply**

**\* @param num2 The second of the two numbers to multiply**

**\* @return the product of the two numbers passed as parameters.**

**\*/**

**public static int multiply(int num1, int num2)**

**{**

**return num1 \* num2;**

**}**

1. Assume the following constant has been declared in a class definition.

**private static final double MIN\_BALANCE = 50.00;**

In addition, the method **meetsMinimum** has been included in the class definition. This method checks to see if the current balance in an account (**accountBalance**) is greater than or equal to the minimum balance as specified in the constant **MIN\_BALANCE**. If so, true is returned, otherwise false is returned.

Method **meetsMinimum** is fully defined below. Do not add any code or alter existing code. Your only task is to write appropriate Javadocs comments for this method in the space provided.

**Write Javadocs here**

**/\*\***

**\* returns true or false depending on whether the current balance**

**\* meets the minimum balance specified**

**\* @return returns true if the current balance meets the minimum**

**\* balance specified; otherwise returns false.**

**\*/**

**public boolean meetsMinimum ()**

**{**

**return accountBalance >= MIN\_BALANCE;**

**}**

1. Which of the following is/are true of the **@param** javadoc tag?
   1. if a method has multiple parameters, then multiple **@param** tags are required
   2. the **@param** tag is used to identify parameters in the method header
   3. both a and b are true
   4. none of the above are true
2. Which of the following is true of black-box testing?
   1. the testing must be done by someone who is knowledgeable about programming
   2. the tester does not need to know how to program
   3. with black-box testing, you can be sure that all possible program paths have been tested
   4. no other testing is necessary when black-box testing is used
3. In \_\_\_\_\_\_ testing, the testing is from the point of view of the user, who does not need to know how to program and does not view the code.
   1. white-box
   2. black-box testing
4. Java provides a framework for testing called
   1. JavaTest
   2. JavaUnitTest
   3. JUnit
   4. JTest

# UML Diagram

1. Draw the UML diagram that represents the **Wizard** class shown here.

**public class Wizard**

**{**

**private String wizardName;**

**private int wizardYear;**

**public Wizard(String wizardName, int wizardYear)**

**{**

**this.wizardName = wizardName;**

**this.wizardYear = wizardYear;**

**}**

**public Wizard()**

**{**

**this("", 0);**

**}**

**public String getWizardName()**

**{**

**return wizardName;**

**}**

**public void setWizardYear(int wizardYear)**

**{**

**this.wizardYear = wizardYear;**

**}**

**@Override**

**public String toString()**

**{**

**return getWizardName() + " " + wizardYear;**

**}**

**}**

**Place UML diagram here**



1. Below is a partial implementation of class **ServiceProvider**. The implementation of the methods is not included, because it is not necessary for this problem. Show how to represent this class using UML notation. Do *not* complete the implementation of the methods.

**public class ServiceProvider**

**{**

**private String providerId;**

**private String providerName;**

**private double[] ratePlans;**

**public ServiceProvider(String providerId, String providerName,**

**double[] ratePlans)**

**{...}**

**public String getProviderId()**

**{...}**

**public String getProviderName()**

**{...}**

**public double maxRatePlan()**

**{...}**

**}**

**ServiceProvider**

**-providerId : String**

**-providerName : String**

**-ratePlans : double[]**

**+ServiceProvider(providerId : String, providerName : String, ratePlans : double[])**

**+getProviderId() : String**

**+getProviderName() : String**

**+maxRatePlan() : double**

1. In a UML diagram, if no visibility indicator is used with an attribute, then the visibility of the attribute is
   1. package
   2. private
   3. protected
   4. public

# Inheritance

1. For class **Wizard**, ***as defined in the previous problem***, add an **equals** method that overrides the **equals** method of the **Object** class. The overriding method should return true only if both the **wizardName** and the **wizardYear** are the same for the two wizards being compared.

**@Override**

**public boolean equals(Object wiz)**

**{**

**return this.wizardName.equals(((Wizard)wiz).wizardName)**

**&& this.wizardYear == ((Wizard) wiz).wizardYear;**

**}**

1. In the previous problem we defined a **Book** class. The class **SciFiBook**, shown below, is a subclass of **Book**. Add the missing code as described here.
   1. Complete the class header line.
   2. Add the code for the constructor. The parameters passed as arguments are used to initialize the private instance variables.
   3. Add the code for **toString**: The **toString** method of **SciFiBook** returns a string consisting of the string returned by the **toString** method of the **Book** class, followed by a single space, followed by the value of **NebulaAwardWinner**.

**public class SciFiBook extends Book**

**{**

**private boolean NebulaAwardWinner;**

**public SciFiBook(String title, String author,**

**int published, boolean NebulaAwardWinner)**

**{**

**super(title, author, published);**

**this.NebulaAwardWinner = NebulaAwardWinner;**

**}**

**@Override**

**public String toString()**

**{**

**return super.toString() + " " + NebulaAwardWinner;**

**}**

**}**

1. The collection of values of the instance variables of an object is called its
   1. behavior
   2. inheritance
   3. state
   4. descriptor
2. When we extend the behavior of a class, the original class is called the subclass
   1. true
   2. false
3. If we add new instance variables and methods to the subclass, the super class cannot use these new variables and methods.
   1. true
   2. false
4. Every Java class automatically extends the \_\_\_\_ class (either directly, or through a class hierarchy).
   1. Collection
   2. Collections
   3. Object
   4. State
5. The constructor in a subclass can invoke the constructor of the superclass using the keyword
   1. call
   2. invoke
   3. supersize
   4. super
6. To determine the type of an object, use the \_\_\_\_\_ operator.
   1. typeof
   2. instanceof
   3. isA
   4. type
7. The methods determine the ­­­­­­­­­­­­­­­\_\_\_ of an object.
   1. behavior
   2. extension
   3. identity
   4. state
8. In order for a class to extend another class, we must be able to say that an instance of the superclass is an instance of the subclass
   1. true
   2. false
9. In Java, a subclass can extend only one superclass.
   1. true
   2. false
10. Some classes in Java do not explicitly extend a class – that is, there is no **extends** clause in the class header. These classes automatically extend the \_\_\_\_ class.
    1. Head
    2. Object
    3. Parent
    4. Super
11. We use the keyword \_\_\_ to indicate that one class is a subclass of another class.
    1. extends
    2. implements
    3. subclasses
    4. identifies
12. The collection of values of the instance variables of an object is called its
    1. behavior
    2. extension
    3. identity
    4. state
13. In Java, a class can extend multiple superclasses.
    1. true
    2. false
14. The behavior of a class is defined by its
    1. methods
    2. attributes
15. A subclass is not allowed to override a method from the superclass.
    1. true
    2. false
16. The “is-a test” says that
    1. an instance of the superclass has an instance of the subclass as a private instance variable
    2. an instance of the subclass has an instance of the superclass as a private instance variable
    3. an instance of the superclass is an instance of the subclass
    4. an instance of the subclass is an instance of the superclass
17. A subclass can directly access the private instance variables of the superclass.
    1. true
    2. false
18. Which of the following is true of protected instance variables?
    1. protected instance variables can be accessed only by subclasses
    2. protected instance variables can be accessed by any class
    3. protected instance variables can be accessed only by classes in the same package
    4. protected instance variables cannot be accessed by any other class
19. When we add new instance variables or methods to a subclass, the superclass
    1. has access to the new instance variables and methods
    2. has no access to the new instance variables and methods
    3. has access to the new instance variables, but not to the new methods
    4. has access to the new methods, but not to the new instance variables
20. In class **Object**, **x.equals(y)** returns true if and only if
    1. the instance variables for the object referenced by **x** have the same value as the instance variables for the object referenced by **y**
    2. **x == y**
    3. either of the above will result in **x.equals(y)** returning true
    4. none of the above are true

# Abstract Classes

1. An abstract class always contains at least one abstract method.
   1. true
   2. false
2. An abstract class cannot be instantiated.
   1. true
   2. false
3. An abstract class always has at least one abstract method.
4. true
5. false
6. Consider the hierarchy of classes shown below. Note that **AbstractStudent** is an abstract class.



* 1. Which of the following statements are legal? Circle the correct answer.

**Person per = new Person();** legal illegal

**Person per = new AbstractStudent();** legal illegal

**Person ug = new Undergraduate();** legal illegal

**Employee emp = new TeachingAssistant();** legal illegal

* 1. Consider the following block of statements. There is currently an error in one of the statements. Use casting to fix *one* of the statements, so that the *pair* of statements will then execute correctly. Do not add any statements. Do not change these statements except for making a single cast. Note that only *one* of the statements below should be changed.

**Person grad = new Graduate();**

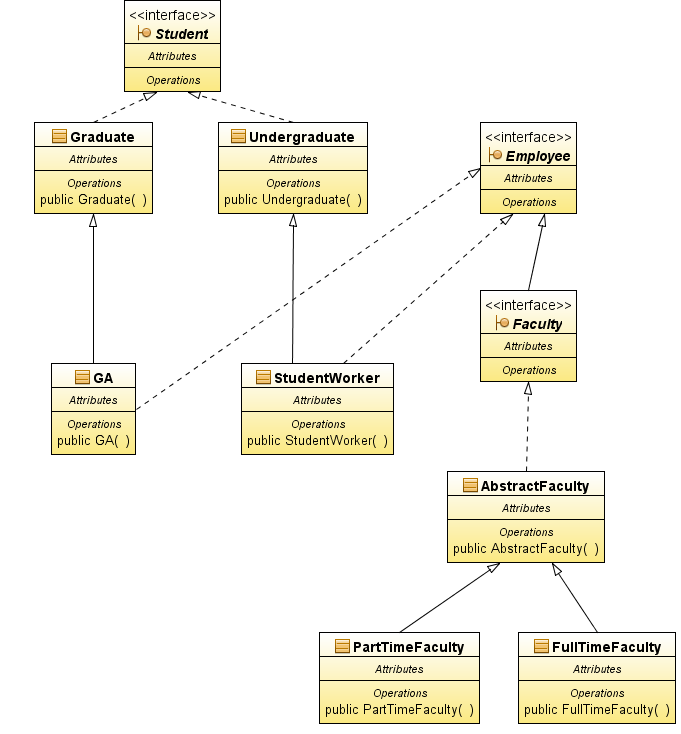
**Graduate grad1 = grad;**

**Rewrite the statement you are correcting here.**

**Graduate grad1 = (Graduate)grad;**

1. This question refers to the UML diagram shown here. For each of the questions on your answer sheet, tell whether it is legal or illegal, and *circle* the correct choice on your answer sheet.

Note that **AbstractFaculty** is an abstract class. All other nodes in this diagram are interfaces or concrete classes.



For each of these statements, tell whether it is legal or illegal. ***Circle*** the correct answer.

**PartTimeFaculty partTimeFac1 = new Faculty(); LEGAL ILLEGAL**

**AbstractFaculty absFac1 = new AbstractFaculty(); LEGAL ILLEGAL**

**AbstractFaculty absFac2 = new FullTimeFaculty(); LEGAL ILLEGAL**

**Employee emp2 = new GA(); LEGAL ILLEGAL**

**GA ga1 = new Graduate(); LEGAL ILLEGAL**

**Graduate grad1 = new Graduate(); LEGAL ILLEGAL**

**Employee emp1 = new Employee(); LEGAL ILLEGAL**

**StudentWorker stuWorker1 = new StudentWorker(); LEGAL ILLEGAL**

**Graduate grad2 = new GA(); LEGAL ILLEGAL**

**Student stu1 = new StudentWorker(); LEGAL ILLEGAL**

1. Assume we have the class hierarchy and the declarations shown below.



**Person person;**

**Employee emp;**

**AbstractStudent absStu;**

**FullTimeEmployee ftEmp;**

**Undergraduate ug;**

**Graduate grad;**

**Professor prof;**

**TeachingAssistant TA;**

Identify each group of statements below as legal or illegal. ***Circle*** the correct response.

**person = new TeachingAssistant(); legal illegal**

**emp = new Professor();**

**prof = emp; legal illegal**

**grad = new TeachingAssistant();**

**TA = (TeachingAssistant) grad; legal illegal**

1. Which of the following is true of abstract classes?
   1. an abstract class cannot be instantiated
   2. an abstract class cannot have subclasses
   3. an abstract class must always have at least one abstract method
   4. a) and c) are true
   5. none of the above is true
2. Which of the following is/are true of abstract classes?
   1. an abstract class cannot be instantiated
   2. an abstract class has at least one abstract method
   3. a class that contains an abstract method must be declared as abstract
   4. all of the above are true
   5. a) and c) only are true
   6. none of the above are true
3. Classes that have all methods fully implemented and are not declared as abstract are referred to as \_\_\_\_\_\_ classes
   1. concrete
   2. complete
   3. full
   4. essential

# Polymorphism

1. The is-a relationship between superclass and subclass allows us to use a subclass object anywhere a superclass object would be allowed. This is referred to as
   1. polymorphic substitution
   2. inheritance
   3. late-binding polymorphism
   4. polymorphic binding
2. The ability to override methods coupled with the ability to determine at runtime which method should be invoked is referred to as
   1. polymorphic substitution
   2. inheritance
   3. late-binding polymorphism
   4. polymorphic binding
3. Define *polymorphic substitution*. Be specific.

***Polymorphic substitution*** refers to the is-a relationship between superclass and subclass that allows us to use a subclass object anywhere a superclass object would be allowed – this means that a reference variable for the supertype may actually store a reference to an instance of the subclass.

1. Fill in the blank:
   1. The ability to override methods coupled with the run-time determination of which method to invoke is referred to as \_\_\_late binding polymorphism\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   2. The is-a relationship between superclass and subclass allows us to use a subclass object anywhere a superclass object would be allowed. This is referred to as

\_\_ polymorphic substitution \_\_\_\_\_.

1. Assume we have the class hierarchy shown below. Each of the classes shown here has its own **toString** method.



In a driver program, suppose we have the following code:

**employees.add(new Employee("Amy", "Smith"));**

**employees.add(new HourlyEmployee("Bill", "Brown", 40, 10.5));**

**employees.add(new SalariedWorker("Susan", "Wright", 80000));**

**for(Employee emp : employees)**

**{**

**System.out.println(emp.toString());**

**}**

When this code is executed, the proper **toString** method is invoked for each employee. This is an example of \_\_\_\_\_late-binding polymorphism \_\_\_\_. (Fill in the answer on your answer sheet.)

# Interfaces

1. Study the UML diagram shown here and answer the questions that follow it. Note that **AbstractClass1** is an abstract class, **Interface1** and **Interface2** are interfaces, and all other nodes in the diagram represent concrete classes.



* 1. Write the class header (the first line of the class, after package and import statements and any class comments) for **Class3**. Hint: The line begins with the words **public class**.

**public class Class3 extends Class2 implements Interface1, Interface2**

* 1. Assume we have made the following declarations in a driver program that uses classes in the hierarchy shown above. All of these declarations are legal.

**Interface1 intface1;**

**Interface2 intface2;**

**AbstractClass1 absClass1;**

**Class1 class1;**

**Class3 class3;**

**Class5 class5;**

Which of the following statements are legal? (Circle the correct choice.)

**intface1 = new Class4(); Legal Not legal**

**intface2 = new Class4(); Legal Not legal**

**intface2 = new Class5(); Legal Not legal**

**intface1 = new Class2(); Legal Not legal**

**absClass1 = new AbstractClass1(); Legal Not legal**

**class5 = new Class3(); Legal Not legal**

**class3 = new Class5(); Legal Not legal**

**absClass1 = new Class3(); Legal Not legal**

**absClass1 = new Class4(); Legal Not legal**

**class1 = new Class4(); Legal Not lega**

1. Assume we have the class hierarchy shown below. For each statement or group of statements following the diagram, tell whether it is illegal or legal.



* 1. **AbstractBook myBook = new AbstractBook();** ILLEGAL
  2. **AbstractBook myBook = new Novel();** LEGAL
  3. **BookInterface myBook = new TextBook();** LEGAL
  4. **SciFiNovel myBook = new Novel();** ILLEGAL
  5. **Novel novel = new Novel();** ILLEGAL

**SciFiNovel sciFi = new SciFiNovel();**

**sciFi = novel;**

* 1. **Novel novel = new SciFiNovel();** LEGAL

**SciFiNovel sciFi = new SciFiNovel();**

**sciFi = (SciFiNovel) novel;**

1. Which of the following is/are true of interfaces
   1. interfaces have no instance variables
   2. all methods in an interface are abstract
   3. both of the above are true
   4. none of the above are true
2. An interface may extend only one other interface.
   1. true
   2. false
3. An object’s type can never be an interface.
   1. true
   2. false
4. Which of the following is/are true?
   1. an interface cannot extend another interface
   2. a class cannot extend an interface
   3. both of the above are true
   4. none of the above are true
5. Which of the following is/are true about interfaces?
   1. interfaces contain only abstract methods
   2. interfaces have no instance variables
   3. both of the above are true
   4. none of the above are true
6. Which of the following is true of interfaces?
   1. all methods are abstract
   2. all methods are public
   3. they have no instance variables
   4. all of the above are true
   5. none of the above is true
7. Suppose **Class1** extends **Class2** and implements **Interface1** and **Interface2**. The correct class header is
   1. **public class Class1 implements Interface1, Interface2 extends Class2**
   2. **public class Class1 extends Class2 implements Interface1 implements Interface2**
   3. **public class Class1 extends Class2 implements Interface1 and Interface2**
   4. **public class Class1 extends Class2 implements Interface1, Interface2**
8. Which of the following are true of interfaces?
   1. all methods are abstract
   2. all methods are public
   3. they have no instance variables
   4. all of the above are true
   5. a) and c) only are true
   6. none of the above are true

# Casting

1. Consider the following classes **Employee** and **HourlyEmployee**.

**public class Employee**

**{**

**private String firstName;**

**private String lastName;**

**private String SSN;**

**public Employee(String firstName, String lastName, String ssn)**

**{**

**this.firstName = firstName;**

**this.lastName = lastName;**

**SSN = ssn;**

**}**

**}**

**public class HourlyEmployee extends Employee**

**{**

**private double hourlyRate;**

**private int hrsWorked;**

**public HourlyEmployee(String firstNameIn, String lastNameIn,**

**String SSNIn, double hourlyRateIn, int hrsWorkedIn)**

**{**

**super(firstNameIn, lastNameIn, SSNIn);**

**hourlyRate = hourlyRateIn;**

**hrsWorked = hrsWorkedIn;**

**}**

**public double calcWeeklySalary()**

**{**

**double overtime = 0.0;**

**if (hrsWorked > 40)**

**{**

**overtime = (hrsWorked - 40) \* (0.5 \* hourlyRate);**

**}**

**return overtime + hrsWorked \* hourlyRate;**

**}**

**}**

Now consider this code that appears in a driver program.

**Employee emp1;**

**HourlyEmployee emp2 = new HourlyEmployee(**

**"Eve", "McDonald", "222", 10.00, 40);**

**emp1 = emp2;**

**double weeklySalary = emp1.calcWeeklySalary();**

The last line of this code results in a syntax error, because **emp1** is declared to be of type **Employee** and **Employee** does not have a method **calcWeeklySalary**. Use a casting operation to rewrite the last line of code so that the program compiles and runs correctly. Do ***not*** change any of the code except for the last line.

**double weeklySalary = ((HourlyEmployee)emp1).calcWeeklySalary();**

1. This statement also refers to the UML diagram. Consider the following code:

**ArrayList<Student> students = new ArrayList<Student>();**

**students.add(new Undergraduate());**

**students.add(new Undergraduate());**

**students.add(new Graduate());**

**Graduate grad = students.get(2);**

**System.out.println(grad);**

This code has an error in the next-to-last line:

**Graduate grad = students.get(2);**

Rewrite the *right-hand side* of the above line to correct the error. You cannot change the left-hand side in any way, nor can you alter any other lines of code.

**Graduate grad = (Graduate) students.get(2);**

# Exceptions

1. Suppose we have a class **Student** containing this method:

**public void addGrade(double gradeIn) throws IllegalArgumentException**

**{**

**if (gradeIn > 100 || gradeIn < 0)**

**{**

**throw new IllegalArgumentException ("grade outside valid range ");**

**} else**

**{**

**quizGrades.add(gradeIn);**

**}**

**}**

A driver class contains this **try-catch** block.

**try**

**{**

**System.out.print("Enter a grade: ");**

**stu1.addGrade(kb.nextInt());**

**} catch (IllegalArgumentException e)**

**{**

**System.out.println("Argument must be between 0 and 100");**

**System.out.println("Please try again");**

**System.out.println(e);**

**}**

Suppose a user runs the program and enters a grade of 200. What is the output?

**Output when user enters 200 in response to the prompt "Enter a grade: "**

**Argument must be between 0 and 100**

**Please try again**

**java.lang.IllegalArgumentException: grade outside valid range**

1. Assume we have an **HourlyEmployee** class with a method **setHourlyRate** as shown here. Modify the body of this method so that an **IllegalArgumentException** is thrown if the parameter **hourlyRate** is less than or equal to 0. The message associated with the exception should be “Hourly rate must be greater than 0”.

**public void setHourlyRate(double hourlyRate)**

**{**

**if (hourlyRate <= 0)**

**{**

**throw new IllegalArgumentException("Hourly rate must " +**

**"be greater than 0");**

**}**

**this.hourlyRate = hourlyRate;**

**}**

1. Rewrite the code for the **setWizardYear** method in the previous problem, so that it throws an **IllegalArgumentException** if the argument is < 1500. The **IllegalArgumentException** thrown should have as its argument the message **Year cannot be before 1500**. The header of the method cannot be changed.

**public void setWizardYear(int wizardYear)**

**{**

**// Place your code here**

**if(wizardYear < 1500)**

**{**

**throw new IllegalArgumentException("Year cannot " +**

**"be before 1500");**

**}**

**this.wizardYear = wizardYear;**

**}**

1. Consider the following code that appears in a driver class for the **Wizard** class defined above in problem 1) and as modified in problem 2).

**Wizard wiz = new Wizard();**

**try**

**{**

**wiz.setWizardYear(1400);**

**}catch(IllegalArgumentException ex)**

**{**

**System.out.println("The wizard year is illegal\n" + ex);**

**}**

**System.out.println("Done");**

Write the output of this program segment in the space below.

**Output**

**The wizard year is illegal**

**java.lang.IllegalArgumentException: Year cannot be before 1500**

**Done**

1. Attempting to read past end of file is an example of a(n)
   1. IllegalArgumentException
   2. EndOfFileException
   3. IOException
   4. RuntimeException
2. Assume we have a class **Student**. One of the methods in class **Student** is shown below.

**public void addGrade(double gradeIn) throws IllegalGradeException**

**{**

**if (gradeIn > 100 || gradeIn < 0)**

**{**

**throw new IllegalGradeException("grade outside valid range ");**

**} else**

**{**

**quizGrades.add(gradeIn);**

**}**

**}**

The following code is in a driver program named **StudentDriver**.

**try**

**{**

**System.out.print("Enter a grade: ");**

**stu1.addGrade(kb.nextInt());**

**} catch (IllegalGradeException e)**

**{**

**// Do nothing**

**}**

* 1. We wish to replace the body of the **catch** clause so that the message **Argument must be between 0 and 100** is displayed. No other message should be displayed from the **catch** clause.

Write the code that should be placed in the body of the **catch** clause here. Do not change any existing code. Do not duplicate code above.

**System.out.println("Argument must be between 0 and 100");**

* 1. We wish to replace the body of the catch clause so that the message that is sent when the argument is thrown is displayed. In the code segment given above from the **Student** class, this would be **grade outside valid range**. However, if the message in the **Student** class is changed, then the message generated by the body of the catch clause must reflect that change. Note that the message **Argument** **must be between 0 and 100** should *not* be displayed.

Write the code that should be placed in the body of the **catch** clause here. Do not change any existing code. Do not duplicate code above.

**System.out.println(e);**

1. Write the complete code for a Java exception class named **RegNumException**. Provide two constructors – a no-arg constructor and a constructor with one **String** parameter.

**public class RegNumException extends Exception**

**{**

**public RegNumException()**

**{**

**super(); // can be omitted**

**}**

**public RegNumException(String msg)**

**{**

**super(msg);**

**}**

**}**

1. Below is the code for the constructor of the **Horse** class. Rewrite the code so that a **RegNumException** is thrown if the value of the parameter **regNum** is less than or equal to 0. When the exception is thrown, the message “Registration number is wrong” is sent as the argument. Rewrite the code for the *entire* constructor, including the method header. Note that **RegNumException** is a checked exception.

**public Horse(int regNum, String name)**

**{**

**this.regNum = regNum;**

**this.name = name;**

**}**

**public Horse(int regNum, String name) throws RegNumException**

**{**

**if(regNum <= 0)**

**{**

**throw new RegNumException("Registration number is wrong");**

**}**

**this.regNum = regNum;**

**this.name = name;**

**}**

1. After modifying the constructor to throw **RegNumException** when the registration number input is less than or equal to 0, we must now catch exceptions when necessary. In particular, the statement

**myHorses.add(new Horse(1, "Sam"));**

in the driver class must be inside a **try** block, with an appropriate **catch** clause following it. Write the code to do this. When catching the exception, print the message **Registration numbers cannot be negative or zero** on one line, followed, on the next line, by the Java-generated exception. Write *only* the **try-catch** block. Do not write any additional code.

**try**

**{**

**myHorses.add(new Horse(1, "Sam"));**

**} catch (RegNumException ex)**

**{**

**System.out.println(**

**"Registration numbers cannot be negative or zero\n" + ex);**

**}**

1. An exception that does not have to be advertised is called
   1. unverified
   2. unblocked
   3. unchecked
   4. unavailable
2. To advertise an exception, include the **throws Exception** clause in the \_\_\_ header.
   1. class
   2. method
   3. package
   4. project
3. Sending a message to a null object is an example of a(n)
   1. **IOException**
   2. **RuntimeException**
4. If a method throws an IOException, the exception must be caught or advertised. To advertise the exception, add \_\_\_ to the method header.
   1. **throw new IOException**
   2. **throws IOException**
   3. **catch IOException**
   4. it is not necessary to add anything to the method header
5. Which of the following is true of **try-catch** blocks?
   1. a **try** block can be followed by multiple **catch**es
   2. if a **try** block is followed by multiple **catch**es, then more than one **catch** may be executed
   3. if a **try** block is followed by multiple **catch**es, then the order of the **catch**es is always irrelevant
   4. all of the above are true
   5. none of the above is true
6. To advertise an exception use the keyword
   1. throw
   2. throws
   3. extends
   4. @throw
7. A try block can only have a single associated catch block.
   1. true
   2. false

# Sorting

1. Consider the following class **Wizard**. Study the class carefully and then answer the question following it. ***Note that this class is slightly different from the earlier implementation.***

**public class Wizard implements Comparable<Wizard>**

**{**

**private String wizardName;**

**private int wizardYear;**

**public Wizard(String wizardName, int wizardYear)**

**{**

**this.wizardName = wizardName;**

**this.wizardYear = wizardYear;**

**}**

**@Override**

**public String toString()**

**{**

**return wizardName + " " + getWizardYear();**

**}**

**@Override**

**public int compareTo(Wizard otherWizard)**

**{**

**return wizardName.compareTo(otherWizard.wizardName);**

**}**

**public int getWizardYear()**

**{**

**return wizardYear;**

**}**

**}**

You will now write part of a driver program. If the driver program is completed successfully, the output shown below should be produced if the given data file is used for input. Note that your program must work for any valid data file.

**Input file (wizards.txt)**

**Dumbledore 1960**

**Snape 1980**

**Harry 2006**

**Trelawney 1990**

**Ron 2006**

**Hermione 2006**

Note that in the last line of output, Harry, Hermione, and Ron all have the same **wizardYear**, so they may appear in any order.

**Output**

**[Dumbledore 1960, Snape 1980, Harry 2006, Trelawney 1990, Ron 2006, Hermione 2006]**

**[Dumbledore 1960, Harry 2006, Hermione 2006, Ron 2006, Snape 1980, Trelawney 1990]**

**[Dumbledore 1960, Snape 1980, Trelawney 1990, Harry 2006, Hermione 2006, Ron 2006]**

You must complete method **main** of the driver program on the next page. Some code is provided for you. ***Do not change any existing code.***  Assume that all necessary import statements have been provided.

**public static void main(String[] args) throws FileNotFoundException**

**{**

**Scanner in = new Scanner(new File("wizards.txt"));**

**ArrayList<Wizard> wizList = new ArrayList<Wizard>();**

**while (in.hasNext())**

**{**

**wizList.add(new Wizard(in.next(), in.nextInt()));**

**}**

**System.out.println(wizList);**

**// Add a single line of code that will sort wizList using the**

**// natural ordering for Wizards.(6 points)**

Add code here

**Collections.sort(wizList);**

**System.out.println(wizList);**

**// Add the necessary code so that the list of Wizards will be**

**// sorted by wizardYear. If two wizards have the same**

Add code here

**// wizardYear, they can be in any order. (14 points)**

**Collections.sort(wizList, new Comparator<Wizard>()**

**{**

**public int compare(Wizard wiz1, Wizard wiz2)**

**{**

**if (wiz1.getWizardYear() > wiz2.getWizardYear())**

**{**

**return 1;**

**} else if (wiz1.getWizardYear() == wiz2.getWizardYear())**

**{**

**return 0;**

**} else**

**{**

**return -1;**

**}**

**}**

**});**

**System.out.println(wizList);**

**} // end main**

1. Suppose we have the class **Book** defined as shown here:

**public class Book implements Comparable<Book>**

**{**

**private String title;**

**private String author;**

**private int published; // year of publication**

**public Book(String title, String author, int published)**

**{**

**this.title = title;**

**this.author = author;**

**this.published = published;**

**}**

**public int getPublished()**

**{**

**return published;**

**}**

**@Override**

**public int compareTo(Book otherBook)**

**{**

**return 0;**

**}**

**public String toString()**

**{**

**return String.format("%-50s %-20s %4d",**

**title, author, published);**

**}**

**}**

* 1. The **compareTo** method has a stub for its body. In the space below, show what code should replace the stub, so that the natural ordering for books is in ascending order by title.

**return title.compareTo(otherBook.title);**

* 1. Suppose we want to add the **equals** method to the **Book** class. We will consider two books to be equal if they have the same title and the same year of publication. Write the Java code to implement this method. Your method must override the **equals** method provided by the **Object** class, which has prototype

**public boolean equals(**[**Object**](file:///C:\Users\merry\Downloads\Java\jdk-6u25-fcs-bin-b04-apidocs-04_Apr_2011\docs\api\java\lang\Object.html)**obj)**

Write the full method, including the complete method header.

**@Override**

**public boolean equals (Object other)**

**{**

**Book otherBook = (Book) other;**

**return this.title.equals(otherBook.title) &&**

**this.published == otherBook.published;**

**}**

* 1. Assume we write a driver that contains an array list of **Book** objects defined as

**ArrayList<Book> bookList = new ArrayList<Book>();**

Also assume that we have filled **bookList** with **Book** objects.

Write a single Java statement that will sort the books in **bookList** according to the natural order for the **Book** class.

**Collections.sort(bookList);**

* 1. Write Java code that could be included in the driver program described in part c) to sort the **Book** objects in **bookList** in ascending order according to the year of publication.

**Collections.sort(bookList, new Comparator<Book>()**

**{**

**@Override**

**public int compare(Book book1, Book book2)**

**{**

**if (book1.getPublished() < book2.getPublished())**

**{**

**return -1;**

**} else if (book1.getPublished() > book2.getPublished())**

**{**

**return 1;**

**} else**

**{**

**return 0;**

**}**

**}**

**});**

1. The **Item** class is defined as shown below.

**public class Item implements Comparable<Item>**

**{**

**private String itemID;**

**private String itemDesc;**

**private double unitPrice;**

**public Item(String itemID, String itemDesc, double unitPrice)**

**{**

**this.itemID = itemID;**

**this.itemDesc = itemDesc;**

**this.unitPrice = unitPrice;**

**}**

**public double getUnitPrice()**

**{**

**return unitPrice;**

**}**

**@Override**

**public String toString()**

**{**

**return itemID + " " + itemDesc + ": unit price = $" + unitPrice;**

**}**

**@Override**

**public int compareTo(Item otherItem)**

**{**

**return this.itemID.compareTo(otherItem.itemID);**

**}**

**@Override**

**public boolean equals (Object obj)**

**{**

**return (this.compareTo((Item)obj) == 0);**

**}**

**}**

* 1. Complete the definition for the **compareTo** method, assuming that items are compared on **itemID**.
  2. Complete the definition for the **equals** method. Your implementation should follow Java’s recommendation that the natural ordering for a class be consistent with **equals**, that is, for non-null instances **e1** and **e2**, **e1.compareTo(e2) == 0** should have the same boolean value as **e1.equals(e2)**.

If you change the **compareTo** method so that items are compared on a different field, you should not have to rewrite the **equals** method.

You may assume that all values being compared are non-null and are of the correct type, so you do not have to check for this in the code you write.

* 1. Suppose we have a driver program with the code shown below. Add one statement at the end of the code supplied here which will result in the items in **itemList** being sorted according to the natural order of the **Item** class. Use the built-in sort method from the **Collections** class to do this.

**ArrayList<Item> itemList = new ArrayList<Item>();**

**... items are added to itemList (do NOT write the code to do this)...**

**// Write a statement which will sort the items in itemList**

**// according to the natural order of the item class.**

**Collections.sort(itemList);**

* 1. Write additional code for the driver class to sort the items in **itemList** in ascending order by **unitPrice**. Use the built-in sort method from the **Collections** class to do this.

**Collections.sort(itemList, new Comparator<Item>()**

**{**

**@Override**

**public int compare(Item item1, Item item2)**

**{**

**if(item1.getUnitPrice() < item2.getUnitPrice())**

**{**

**return -1;**

**} else if(item1.getUnitPrice() > item2.getUnitPrice())**

**{**

**return 1;**

**} else**

**{**

**return 0;**

**}**

**}**

**});**

1. Assume we have a **Horse** class defined as shown below. To save space, the code is not included, but the purpose of each method should be clear. The **compareTo** method compares **Horse** objects by their registration number (stored in **regNum**).

**public class Horse implements Comparable<Horse>**

**{**

**private int regNum;**

**private String name;**

**public Horse(int regNum, String name)**

**{...}**

**public String getName()**

**{...}**

**public int compareTo(Horse horse)**

**{...}**

**@Override**

**public String toString()**

**{...}**

**}**

Assume we have a driver class containing the following code:

**ArrayList<Horse> myHorses = new ArrayList<Horse>();**

**myHorses.add(new Horse(1, "Sam"));**

**myHorses.add(new Horse(5, "Sherman"));**

**...**

Note that many more horses than the two shown here may be added to the array list.

* 1. Write a single Java statement that will sort the horses in the array list according to their natural order.

**Collections.sort(myHorses);**

* 1. Write Java code that can be included in the driver program and will sort the horses according to a new comparator. The new comparator must be defined to sort the horses by name. The code for the new comparator and the command to sort the horses using the new comparator must be included in the driver class. That is, you must write the comparator as an inner class, rather than a separate class.

**Collections.sort(myHorses, new Comparator<Horse>()**

**{**

**public int compare(Horse horse1, Horse horse2)**

**{**

**return horse1.getName().compareTo(horse2.getName());**

**}**

**});**

1. This problem uses the **Student** class from the preceding problem. Shown below is part of a driver for the **Student** class. Some pieces of code are missing from the driver. Fill in the missing parts, as indicated below.

**public class StudentDriver**

**{**

**public static void main(String[] args)**

**{**

**ArrayList<Student> myStudents = new ArrayList<Student>();**

**myStudents.add(new Student("999", "Alice", "Li"));**

**myStudents.add(new Student("333", "John", "Smith"));**

**myStudents.add(new Student("111", "Amy", "Smith"));**

**In this text box, write a single Java statement that uses the Java sort method to sort the students in their natural order. Your statement must work for any array list of valid student records.**

**Collections.sort(myStudents);**

**In this text box, write Java code that uses the Java sort method to sort the students by last name. Students with the same last name must be sorted by first name. Your statement must work for any array list of valid student records.**

**Collections.sort(myStudents, new Comparator<Student>()**

**{**

**public int compare(Student stu1, Student stu2)**

**{**

**if (stu1.getLastName().equals(stu2.getLastName()))**

**{**

**return stu1.getFirstName().compareTo(stu2.getFirstName());**

**} else**

**{**

**return stu1.getLastName().compareTo(stu2.getLastName());**

**}**

**}**

**});**

**}**

**}**

1. Consider the following class **Student**. This class is completely defined except for the implementation of **compareTo**. Write the implementation of this method in the space provided below. Read the comment preceding the method to see how it should be implemented.

**import java.util.ArrayList;**

**public class Student implements Comparable<Student>**

**{**

**private String stuId;**

**private String firstName;**

**private String lastName;**

**public Student(String stuId, String firstName,**

**String lastName)**

**{**

**this.stuId = stuId;**

**this.firstName = firstName;**

**this.lastName = lastName;**

**}**

**public String getFirstName()**

**{**

**return firstName;**

**}**

**public String getLastName()**

**{**

**return lastName;**

**}**

**@Override**

**public String toString()**

**{**

**return stuId + " " + getFirstName() + " " + getLastName();**

**}**

**// This method compares two students based on the String**

**// comparison of the student ids (stuId).**

**@Override**

**public int compareTo(Student stu)**

**{**

**Insert code for this method inside this text box.**

**return stuId.compareTo(stu.stuId);**

**}**

**}**

1. To override the natural ordering for a class, use the \_\_\_\_\_\_ interface.
   1. Collection
   2. Comparable
   3. Comparator
   4. Sort

# Recursion

1. We have a method **mystery** defined as follows:

**public static int mystery(int numIn)**

**{**

**if(numIn == 0)**

**{**

**return 0;**

**}else**

**{**

**if ((numIn % 10) % 2 == 0)**

**{**

**return numIn % 10 + mystery(numIn / 10);**

**} else**

**{**

**return mystery(numIn / 10);**

**}**

**}**

**}**

* 1. **mystery(8342) = \_\_\_\_\_\_\_14\_\_\_\_\_\_\_\_\_\_\_**
  2. **mystery(1357) = \_\_\_\_\_\_\_0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. Assume method **mystery** is defined as shown here.

**public static int mystery(int numIn)**

**{**

**if(numIn == 0)**

**{**

**return 1;**

**} else**

**{**

**if ((numIn / 10) % 2 == 0)**

**{**

**return numIn / 10 - mystery(numIn / 10);**

**} else**

**{**

**return mystery(numIn / 10);**

**}**

**}**

**}**

* 1. **mystery(8342) = \_\_\_\_\_\_825\_\_\_\_\_\_\_\_\_\_\_\_**
  2. **mystery(1357) = \_\_\_\_\_\_\_-1\_\_\_\_\_\_\_\_\_\_\_**

1. Find the output of the following program.

**public class RecursionExercise**

**{**

**public static void main(String[] args) throws FileNotFoundException**

**{**

**System.out.println(mystery("Hello"));**

**System.out.println(**

**mystery("Spring Break is soon!"));**

**}**

**public static String mystery(String str)**

**{**

**if (!str.contains(" "))**

**{**

**return rw(str);**

**} else**

**{**

**return rw(str.substring(0, str.indexOf(" "))) + " " +**

**mystery(str.substring(str.indexOf(" ") + 1));**

**}**

**}**

**public static String rw(String str)**

**{**

**String rStr = "";**

**for(int i = str.length() - 1; i >= 0; i--)**

**{**

**rStr += str.charAt(i);**

**}**

**return rStr;**

**}**

**OUTPUT**

**olleH**

**gnirpS kaerB si !noos**

**}**

1. Find the output of the following:

**public static void main(String[] args)**

**{**

**System.out.println(mystery("Bearcats"));**

**System.out.println(mystery("Northwest Bearcats Defeat Fort Hays"));**

**}**

**public static String mystery(String str)**

**{**

**if (!str.contains(" "))**

**{**

**return " DONE";**

**} else**

**{**

**return str.substring(str.indexOf(" ") + 1, str.indexOf(" ") + 2) +**

**mystery(str.substring(str.indexOf(" ") + 1));**

**}**

**}**

**WRITE OUTPUT HERE**

**DONE**

**BDFH DONE**

1. Euclid’s algorithm, for computing the greatest common divisor, is given here:

**gcd(a, b) = gcd(b, a modulo b) if b > 0;**

**gcd(a, 0) = a.**

Show how to use Euclid’s algorithm to find the greatest common divisor of 8 and 17 by completing the following. You must show EVERY step. Note that you will *not* write any Java code for this problem.

**gcd(8, 17) = gcd(17, 8) = gcd(8, 1) = gcd(1, 0) = 1**

1. Find the output:

**public static void main(String[] args)**

**{**

**mystery(1234);**

**OUTPUT**

**420531**

**System.out.println();**

**mystery(12345);**

**System.out.println();**

**}**

**public static void mystery (long n)**

**{**

**System.out.print(n % 10);**

**if ((n / 10) != 0)**

**{**

**n /= 10;**

**mystery(n / 10);**

**}**

**}**

1. A method that invokes itself is said to be
   1. iterative
   2. recursive
   3. self-invoking
   4. none of the above

# Java Collections Framework

1. A toString method for printing collections is implemented in the
   1. Collection class
   2. AbstractCollection class
   3. ConcreteCollection class
   4. none of the above
2. An Iterator object has methods
   1. hasPrevious
   2. previous
   3. next
   4. all of the above
3. Assume we have declared two **String** values. The first string value is stored in a variable named **regex** and represents a regular expression. The second string value is stored in a variable named **str**. Write Java statements that will return **true** if the value stored in **str** matches the pattern represented by **regex** and **false** otherwise. Do not write an entire program or method. Write *only*  the statements necessary to perform the task described. Do not include any import statements – you may assume that whatever import statements are needed will be provided.

Example: For the declarations shown here

**String regex = "[a-z]+";**

**String str = "happy halloween";**

your code should return **false**.

For the declarations shown here:

**String regex = "[a-z]+";**

**String str = "halloween";**

your code should return **true**.

REMEMBER: Your code must work for any valid regular expressions stored in **regex** and any valid string stored in **str**.

**return Pattern.matches(regex, str);**

**OR**

**Pattern pattern = Pattern.compile(regex);**

**Matcher matcher = pattern.matcher(str);**

**return matcher.matches();**

1. The root interface in the Java Collections Framework is the \_\_\_\_ interface.
   1. Collection

# Linked Lists

1. Find the output of the following program segment.

**LinkedList<Integer> listOfInts = new LinkedList<Integer>();**

**listOfInts.add(15);**

**OUTPUT**

**[78, 15, 93, 2, 74]**

**listOfInts.add(93);**

**listOfInts.add(74);**

**listOfInts.addFirst(90);**

**listOfInts.add(3, 2);**

**listOfInts.addFirst(78);**

**listOfInts.remove(1);**

**System.out.println(listOfInts);**

1. Linked lists do not allow for direct access.
   1. true
   2. false
2. Find the output of the program segment given below.

**LinkedList<Integer> myList = new LinkedList<Integer>();**

**myList.addFirst(14);**

**OUTPUT**

**My List: [27, 14, 80, 75, 19, 33, 100]**

**myList.addFirst(27);**

**myList.addLast(19);**

**myList.add(33);**

**myList.addLast(100);**

**myList.add(myList.indexOf(14) + 1, 80);**

**myList.add(3, 75);**

**System.out.println("My List: " + myList);**

1. Find the output of the following code segment:

**LinkedList<Integer> myList =**

**new LinkedList<Integer>();**

**myList.add(10);**

**myList.add(50);**

**myList.add(60);**

**myList.add(20);**

**myList.add(100);**

**OUTPUT**

**in first loop: 10**

**in first loop: 60**

**in first loop: 80**

**in second loop: 100**

**in second loop: 160**

**in second loop: 210**

**in second loop: 220**

**final sum = 220**

**myList.add(20);**

**myList.add(10);**

**ListIterator<Integer> iter =**

**myList.listIterator();**

**int sum = 0;**

**boolean done = false;**

**while(iter.hasNext() && !done)**

**{**

**int num = iter.next();**

**if(num <= 50)**

**{**

**sum += num;**

**System.out.println(**

**"in first loop: " + sum);**

**}**

**if(sum > 75)**

**{**

**done = true;**

**}**

**}**

**while(iter.hasPrevious())**

**{**

**int num = iter.previous();**

**sum += num;**

**System.out.println(**

**"in second loop: " + sum);**

**}**

**System.out.println("final sum = " + sum);**

1. Find the output of the following program segment:

**LinkedList<Integer> myInts = new LinkedList<Integer>();**

**myInts.add(80);**

**myInts.add(90);**

**OUTPUT**

**90**

**6**

**[14, 80, 90, 2, 35, 100, 75, 50]**

**[14, 90, 2, 35, 100, 75, 50]**

**myInts.add(35);**

**myInts.add(100);**

**myInts.add(75);**

**myInts.addFirst(14);**

**myInts.add(3, 2);**

**myInts.get(1);**

**myInts.add(50);**

**System.out.println(myInts.get(2));**

**System.out.println(myInts.indexOf(75));**

**System.out.println(myInts);**

**myInts.remove(1);**

**System.out.println(myInts);**

1. Find the output of the following code segment:

**LinkedList<Integer> myList = new LinkedList<Integer>();**

**myList.addFirst(23);**

**OUTPUT**

**[34, 23, 100, 1, 95, 18]**

**myList.addLast(1);**

**myList.addLast(18);**

**myList.addFirst(34);**

**myList.add(2, 100);**

**myList.add(myList.indexOf(1) + 1, 55);**

**myList.set(4, 95);**

**System.out.println(myList);**

1. Assume we have some integer values stored in a **LinkedList** named **myList**. Write a code segment to print out all values in the list that are indexed by an even number. Values should be printed on a single line, with pairs of values separated by a single space.

Example: If your list is **14 75 90 30 27 18 33**, then the output of your code segment should be **14 90 27 33**.

Your code must work for a linked list of any size, and containing an arbitrary list of integers.

Write only a code segment; do not include any class or method headers.

**for(int i = 0; i < myList.size(); i+=2)**

**{**

**System.out.print(myList.get(i) + " ");**

**}**

1. Linked lists do not allow for direct access.
   1. true
   2. false
2. In Java, linked lists are implemented as singly-linked lists.
   1. true
   2. false

# Stacks

1. Find the output of the following program segment.

**PureStack<String> myHorses = new PureStack<String>();**

**myHorses.push("Eddie");**

**myHorses.push("Trouble");**

**OUTPUT**

**Sherman**

**Sam Daisy Trouble Eddie**

**myHorses.push("Ross");**

**myHorses.pop();**

**myHorses.push("Daisy");**

**myHorses.push("Sam");**

**myHorses.push("Sherman");**

**System.out.println(**

**myHorses.peek());**

**myHorses.pop();**

**while (!myHorses.isEmpty())**

**{**

**System.out.print(myHorses.pop() + " ");**

**}**

1. \_\_\_\_\_ is an example of a LIFO object.
   1. linked list
   2. stack
   3. queue
   4. tree
2. The pop operation for a stack
   1. adds an element to the top of the stack
   2. removes the element at the top of the stack, but does not return it
   3. removes and returns the element at the top of the stack
   4. returns the element at the top of the stack, but does not remove it
3. The Stack class extends
   1. the Vector class
   2. the ArrayList class
   3. the AbstractStack class
   4. the LinkedList class
4. Find the output of the program segment given below.

**OUTPUT**

**Peeking : Zelda**

**My Stack: Carmen Eve Zelda Leonard Midge**

**Stack<String> myStack =**

**new Stack<String>();**

**myStack.push("Midge");**

**myStack.push("Leonard");**

**myStack.push("Zelda");**

**System.out.println(**

**"Peeking : " + myStack.peek());**

**myStack.push("Eve");**

**myStack.push("Carmen");**

**System.out.print("My Stack: ");**

**while(!myStack.isEmpty())**

**{**

**System.out.print(**

**myStack.pop() + " ");**

**}**

**System.out.println();**

1. Find the output of the following program segment. **PureStack** is identical to the class created in the lab activity and is implemented using a **Vector** as the underlying data structure. The **toString** method of the **PureStack** class calls the **toString** method of the **Vector** class.

**PureStack<Integer> myStack = new PureStack<Integer>();**

**myStack.push(100);**

**OUTPUT**

**75**

**[100, 15]**

**myStack.push(15);**

**myStack.push(75);**

**System.out.println(myStack.peek());**

**myStack.pop();**

**System.out.println(myStack);**

1. A stack is an example of a \_\_\_\_\_ structure.
   1. FIFO
   2. FOFO
   3. LIFO
   4. LOLO
2. The stack operation **peek** retrieves *and* removes the top element of the stack.
   1. true
   2. false

# Queues

1. Find the output of the program segment given below. Assume we have implemented a class **Queue**, with methods **add** for enqueue and **remove** for dequeue.

**Queue<Integer> myQ = new Queue<Integer>();**

**myQ.add(35);**

**myQ.add(75);**

**System.out.println("Front: " + myQ.front());**

**myQ.add(105);**

**myQ.add(16);**

**myQ.remove();**

**myQ.add(88);**

**System.out.print("My Q: ");**

**while (!myQ.isEmpty())**

**{**

**System.out.print(myQ.remove() + " ");**

**}**

**OUTPUT**

**Front: 35**

**My Q: 75 105 16 88**

1. Find the output of the following program segment. **PureQueue** is an implementation of the queue data structure, using a **LinkedList** as the underlying data structure. The **toString** method of the **PureQueue** class returns each element of the queue on a separate line, beginning with the element at the front of the queue.

**PureQueue<Integer> myQ = new PureQueue<Integer>();**

**myQ.enqueue(100);**

**OUTPUT**

**100**

**74**

**14**

**90**

**myQ.enqueue(74);**

**myQ.enqueue(14);**

**System.out.println(myQ.front());**

**myQ.enqueue(90);**

**myQ.dequeue();**

**System.out.println(myQ);**

1. Supply the missing code for the **enqueue** and **dequeue** methods..

**import java.util.LinkedList;**

**public class Queue<E>**

**{**

**private LinkedList<E> queue;**

**/\*\***

**\* Constructor – DO NOT ALTER THE CODE FOR THE CONSTRUCTOR.**

**\*/**

**public Queue()**

**{**

**queue = new LinkedList<E>();**

**}**

**/\*\***

**\* DO NOT ALTER THE CODE FOR THIS METHOD.**

**\* @return true if the queue is empty, false otherwise.**

**\*/**

**public boolean isEmpty()**

**{**

**return queue.isEmpty();**

**}**

**/\*\***

**\* Add an element to the end of the queue.**

**\* @param element The element to be added.**

**\*/**

**public void enqueue(E element)**

**{// ADD THE CODE FOR THIS METHOD.**

**// queue.addLast(element);**

**// OR**

**queue.add(element);**

**}**

**/\*\***

**\* Removes and returns the element at the front of the queue.**

**\* @return the element at the front of the queue.**

**\* @throws EmptyQueueException, with message “Queue is empty”**

**\* if the queue is empty**

**\*/**

**public E dequeue()**

**{// ADD THE CODE FOR THIS METHOD.**

**if(queue.isEmpty())**

**{**

**throw new EmptyQueueException("Queue is empty");**

**}**

**//return queue.removeFirst();**

**// OR**

**return queue.remove();**

**}**

1. A queue is an example of a \_\_\_\_\_ structure.
   1. FIFO
   2. FOFO
   3. LIFO
   4. LOLO

# Hashing

1. Suppose we are using the hash function **h(key) = key MOD 7** to store records with keys **13**, **34**, **15**, **23**, **35**, **27**.
   1. Show where each key is stored in the table below. Use linear probing to resolve collisions. You must apply h to the keys in the order in which they are listed.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **34** | **15** | **23** | **35** | **27** |  | **13** |

* 1. Show where each key is stored in the table below. Use chaining to resolve collisions. You must apply h to the keys in the order in which they are listed.

|  |  |
| --- | --- |
| **Index** | **list of values** |
| **0** | **list 🡪 35 🡪 null** |
| **1** | **list 🡪 15 🡪 null** |
| **2** | **list 🡪 23 🡪 null** |
| **3** |  |
| **4** |  |
| **5** |  |
| **6** | **list 🡪 13 🡪 34 🡪 27 🡪 null** |

1. How is the position of an element in a hash table determined?
   1. It depends on when it is added to the list. The first element is in position 0, the second element in position 1, etc.
   2. It depends on the value of the key. The key must be numerical and the address represented by the key is used as the storage location.
   3. A function is applied to the key value. The result returned by the function determines the position of the element.
   4. A random number generator is used, with the key value as the seed. The number generated is used as an address for the storage location of the element.
2. When hashing is used, search time is
   1. O(1)
   2. O(log n)
   3. O(n log n)
   4. O(n)
3. Suppose we are using the hash function **h(key) = key MOD 8** to store records with keys **11, 18, 26, 39, 15**, and **80**. Show where each key is stored in the table. Use linear probing to resolve collisions. You must apply **h** to the keys in the order in which they are listed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **15** | **80** | **18** | **11** | **26** |  |  | **39** |

1. A rule of thumb for the size of the hash table is:
   1. the hash table should be one-half the size of the data set (50%)
   2. the hash table should be the same size as the data set (100%)
   3. the hash table should be one and one-half times the size of the data set (150%)
   4. the hash table should be 3 times the size of the data set (300%)
2. Which of the following is/are true of hashing?
   1. elements are not retrieved in sorted order
   2. performance may be slowed down if you insert records frequently (resulting in frequent dynamic resizing of the hash table)
   3. both of the above are true
   4. none of the above
3. Suppose we are using the hash function **h(key) = key MOD 10** to store records with keys **32, 55, 15, 98, 42**, **13, 22, 89, 99.** Show where each key is stored in the table below. Use linear probing to resolve collisions. You must apply **h** to the keys in the order in which they are listed.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **99** |  | **32** | **42** | **13** | **55** | **15** | **22** | **98** | **89** |

1. Suppose we are using the hash function **h(key) = key MOD 7** to store records with keys **11, 18, 20, 27, 39, 15**, and **14**. Show where each key is stored in the table. Use linear probing to resolve collisions. You must apply **h** to the keys in the order in which they are listed.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **27** | **39** | **15** | **14** | **11** | **18** | **20** |

1. Suppose we are using the hash function **h(key) = key MOD 10** to store records with keys **32, 55, 15, 98, 42**, **13, 22, 89, 99.** Show where each key is stored in the table below. Use chaining to resolve collisions. You must apply **h** to the keys in the order in which they are listed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** |  |  |  |  |  |  |  |
| **0** |  |  |  |  |  |  |  |
| **1** |  |  |  |  |  |  |  |
| **2** |  | **🡪** | **32** | **🡪** | **42** | **🡪** | **22** |
| **3** |  | **🡪** | **13** |  |  |  |  |
| **4** |  |  |  |  |  |  |  |
| **5** |  | **🡪** | **55** | **🡪** | **15** |  |  |
| **6** |  |  |  |  |  |  |  |
| **7** |  |  |  |  |  |  |  |
| **8** |  | **🡪** | **98** |  |  |  |  |
| **9** |  | **🡪** | **89** | **🡪** | **99** |  |  |

1. Suppose we are using the hash function **h(key) = key MOD 10** to store records with keys **12, 22, 33, 32, 42, 80, 89, 29, 110, 36**. Show where each key is stored in the table. Use linear probing to resolve collisions. You must apply **h** to the keys in the order in which they are listed.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **80** | **29** | **12** | **22** | **33** | **32** | **42** | **110** | **36** | **89** |

1. Repeat the problem above, but this time store values in the table and use chaining to resolve collisions.

|  |  |
| --- | --- |
| **Index** |  |
| **0** | **list 🡪 80 🡪 110 🡪 null** |
| **1** |  |
| **2** | **list 🡪 12 🡪 22 🡪 32 🡪 42 🡪 null** |
| **3** | **list 🡪 33 🡪 null** |
| **4** |  |
| **5** |  |
| **6** | **list 🡪 36 🡪 null** |
| **7** |  |
| **8** |  |
| **9** | **list 🡪 89 🡪 29 🡪 null** |

1. If two elements hash to the same value, we say a(n) \_\_\_\_ has occurred.
   1. accident
   2. wreck
   3. crash
   4. collision

# Sets and Maps

1. Consider the following class

**public class MyMap**

**{**

**TreeMap<String, LinkedList<Integer>> myMap;**

**public MyMap()**

**{**

**myMap = new TreeMap<String, LinkedList<Integer>>();**

**}**

**public void addString(String str)**

**{**

**myMap.put(str, new LinkedList<Integer>());**

**}**

**public void addInteger(String str, Integer num)**

**{**

**LinkedList<Integer> myList = myMap.get(str);**

**myList.add(num);**

**myMap.put(str, myList);**

**}**

**public double myMethod(String myStr)**

**{**

**double returnVal = 0.0;**

**for (String str : myMap.keySet())**

**{**

**if (str.equals(myStr))**

**{**

**double sum = 0.0;**

**LinkedList<Integer> myList = myMap.get(str);**

**if (myList.size() == 0)**

**{**

**return -1;**

**} else**

**{**

**for (int num : myList)**

**{**

**sum += num;**

**}**

**}**

**returnVal = sum / myList.size();**

**}**

**}**

**return returnVal;**

**}**

**}**

Below is a segment of code from a driver program that uses this class. Find the output of the code segment.

**MyMap myMap = new MyMap();**

**OUTPUT**

**30.0**

**9.0**

**myMap.addString("Midge");**

**myMap.addString("Leonard");**

**myMap.addInteger("Midge", 10);**

**myMap.addInteger("Midge", 20);**

**myMap.addInteger("Midge", 60);**

**myMap.addInteger("Leonard", 5);**

**myMap.addInteger("Leonard", 1);**

**myMap.addInteger("Leonard", 10);**

**myMap.addInteger("Leonard", 20);**

**System.out.println(myMap.myMethod("Midge"));**

**System.out.println(myMap.myMethod("Leonard"));**

1. The Set interface
   1. allows duplicate elements
   2. extends the SortedSet interface
   3. both a and b are true
   4. none of the above
2. If you are implementing a map and you want a guarantee that keys will be in order (using either the natural order, or the order supplied by a comparator), which of the following should you use?
   1. HashMap
   2. TreeMap
3. Consider this partially completed class. Imports are not included, but assume you have all the imports you need. The tree map is used to store mappings where keys are types of animals (Dog, Horse, Whale, for example). The values are names of animals of that type.

**public class Animals**

**{**

**private TreeMap<String, LinkedList<String>> animals;**

**public Animals()**

**{**

**}**

**public void addAnimal(String animalType, String animalName)**

**{**

**}**

**public LinkedList<String> getNames(String animalType)**

**{**

**}**

**@Override**

**public String toString()**

**{**

**return animals.toString();**

**}**

**}**

When this class is completed, the following code might appear in a driver program:

**Animals myAnimals = new Animals();**

**myAnimals.addAnimal("Dog", "Zelda");**

**myAnimals.addAnimal("Dog", "Midge");**

**myAnimals.addAnimal("Horse", "Sherman");**

**myAnimals.addAnimal("Dog", "Leonard");**

**System.out.println(myAnimals.getNames("Dog"));**

**System.out.println(myAnimals.getNames("Horse"));**

**System.out.println(myAnimals.getNames("Elephant"));**

**System.out.println(myAnimals);**

This code would result in the following output:

**[Zelda, Midge, Leonard]**

**[Sherman]**

**null**

**{Dog=[Zelda, Midge, Leonard], Horse=[Sherman]}**

* 1. Write the code for the constructor for this class. The constructor initializes the private instance variable to a new empty tree map.

**public Animals()**

**{**

**animals = new TreeMap<String, LinkedList<String>>();**

**}**

* 1. Write the code for the method **getNames**. This method has a single **String** parameter representing a type of animal. It returns the values associated with that animal type – that is, a linked list of animal names. See the sample driver code and output above for more clarification. Note that it is permissible for this method to return **null** – again, see the output above for clarification.

**public LinkedList<String> getNames(String animalType)**

**{**

**return animals.get(animalType);**

**}**

* 1. Write the code for method **addAnimal**. This method has two parameters – the first representing an animal type and the second representing the name of an animal of that type. If the animal type already exists in the tree map, you must add the name of the animal to the corresponding linked list. If the animal type does not exist, you must create a new entry in the tree map.

**public void addAnimal(String animalType, String animalName)**

**{**

**LinkedList<String> names = animals.get(animalType);**

**if(names == null)**

**{**

**names = new LinkedList<String>();**

**names.add(animalName);**

**animals.put(animalType, names);**

**} else**

**{**

**names.add(animalName);**

**}**

**}**

1. Study the following code for the **Book** class:

**public class Book**

**{**

**private String title;**

**private String authorName;**

**public Book(String title, String authorName)**

**{**

**this.title = title;**

**this.authorName = authorName;**

**}**

**public String getTitle()**

**{**

**return title;**

**}**

**@Override**

**public String toString()**

**{**

**return String.format("%-30s, %-30s", title, authorName);**

**}**

**}**

Class **Books** is shown below. The **TreeMap** variable **bookMap** is used to map a **String**, representing the ISBN of a book, to a **Book** object. Supply the missing code.

**public class Books**

**{**

**private TreeMap<String, Book> bookMap;**

**public Books()**

**{**

**bookMap = new TreeMap<String, Book>();**

**}**

**// Add a key-value pair; isbn is the key, book is the value associated**

**// with the key.**

**public void addBook(String isbn, Book book)**

**{**

**bookMap.put(isbn, book);**

**}**

**// Given a key (the isbn), return the Book object associated with the**

**// key.**

**public Book getBook(String isbn)**

**{**

**return bookMap.get(isbn);**

**}**

**// Given a key (the isbn), return the title of the Book object**

**// associated with the key.**

**public String getBookTitle(String isbn)**

**{**

**return bookMap.get(isbn).getTitle();**

**}**

**}**

1. Assume we have a class named **Student** with private instance variables:

**private String name; // name of student**

**private String classification; // freshman, sophomore, etc.**

This class has a constructor with two parameters and getter methods for the two instance variables. In addition, we have a class named **ClassList** as shown below.

**public class ClassList**

**{**

**TreeMap<String, LinkedList<Student>> classList;**

**public ClassList()**

**{**

**classList = new TreeMap<String, LinkedList<Student>>();**

**}**

**/\*\***

**\* Adds a student to the tree map classList.**

**\* @param student The student to be added.**

**\*/**

**public void addStudent(Student student)**

**{**

**String classification = student.getClassification();**

**LinkedList<Student> tempList =**

**classList.get(student.getClassification());**

**if(tempList == null)**

**{**

**tempList = new LinkedList<Student>();**

**}**

**tempList.add(student);**

**classList.put(classification, tempList);**

**}**

**}**

The **TreeMap** named **classList** has strings representing classifications (freshman, sophomore, junior, or senior) as keys; the value associated with a key consists of the students who are that classification. For example, suppose we have the following students:

Amy – freshman

Bill – sophomore

Sue – senior

Alex – freshman

Kim – sophomore

If we add these students to the tree map, we will have an entry for keys with values **freshman**, **sophomore**, and **senior**. The value associated with the key **freshman** will be the linked list of students **Amy** and **Alex**. For the key **sophomore**, we have the linked list of students **Bill** and **Kim**. For the key **senior**, we have a linked list of students containing only **Sue**. If these are the only students we have added, we have no entry for the junior classification. Note that your code must work for any classification scheme we want to use. For example, we might want to use this to classify by graduate and undergraduate, rather than freshman, sophomore, junior senior.

**Your task:** Add the code to implement method **addStudent**.

1. Which of the following is/are true of maps?
   1. keys are unique
   2. values associated with keys may not be unique – that is, two different keys may have the same associated value
   3. the **Map** interface extends the **Collection** interface
   4. the only Java class that implements the **Map** interface is **TreeMap**

# Trees, Binary Trees, Binary Search Trees

1. Draw the binary search tree that results from inserting these elements into the tree, using the algorithm discussed in class. Elements must be inserted in the order given here:

42 93 19 10 50 99 12

**Draw Tree Here**



1. Consider the binary search tree shown here. Draw the tree that results when 50 is removed from the tree. You must use the algorithm discussed in class.

**Original Tree**



**Result of Removing 50**



1. Consider the binary search tree shown here. Draw the tree that results from performing a right rotation around 50. You must use the algorithm discussed in class.

**Result of Right Rotation around 50**



**Original Tree**



1. The following questions refer to the tree in the text box on the right. Write the answer in the space provided, or, if possible answers are given, ***circle*** the correct answer.
   1. How many leaves does this tree have? \_\_\_\_\_5\_\_\_\_\_



* 1. Is it a binary tree? YES NO
  2. Is it a binary search tree? YES NO
  3. Is it a complete tree? YES NO
  4. Is it a two-tree? YES NO
  5. Is it a full-tree? YES NO
  6. Is it a heap? YES NO
  7. List the nodes in the order they appear in an in-order traversal.

\_\_G D H B I E J A C F\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. The following questions refer to the tree shown below. Write the answer in the space provided, or, if possible answers are given, ***circle*** the correct answer.



* 1. How many leaves does this tree have? \_\_\_\_\_6\_\_\_\_\_
  2. Is it a binary tree? YES NO
  3. Is it a binary search tree? YES NO
  4. Is it a complete tree? YES NO
  5. Is it a two-tree? YES NO
  6. Is it a full-tree? YES NO
  7. Is it a heap? YES NO
  8. List the nodes in the order they appear in a pre-order traversal.

\_\_50 25 12 10 20 40 35 44 75 60 90\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. A two-tree is a full tree.
   1. true
   2. false
2. A complete tree is a two-tree.
   1. true
   2. false
3. A full true is a two-tree.
   1. true
   2. false
4. A complete tree is a binary search tree.
   1. true
   2. false
5. Refer to this tree to answer the questions below.



* 1. What is the height of the tree? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 4
  2. How many leaves does this tree have? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3
  3. What is the parent of 60? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 75
  4. List the children of 25. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 14, 30
  5. List the descendents of 30. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 26, 28
  6. What is the root of the tree? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 50
  7. How many probes are necessary to find 60? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3
  8. How many probes are necessary to discover that 29 is not in the tree? \_\_\_\_\_\_\_\_\_\_\_\_\_ 6
  9. List the elements in the order in which they would be traversed in an inorder traversal.

**14 25 26 28 30 50 60 65 75**

* 1. List the elements in the order in which they would be traversed in a preorder traversal.

**50 25 14 30 26 28 75 60 65**

1. Draw the binary search tree that results when you insert these elements into the tree, in the order specified here:  **18 22 95 4 12 19 76**

**DRAW TREE HERE**



1. Remove node 26 from the binary search tree shown here, and draw the resulting binary search tree.





1. Remove node 25 from the binary search tree shown here, and draw the resulting binary search tree.





1. For the binary search tree shown below, rotate right around 75.





1. For the binary search tree shown below, rotate left around 25.



1. Is the tree shown below
   1. a two-tree? YES NO
   2. a full tree? YES NO
   3. a complete tree? YES NO



1. Is the tree shown below
   1. a two-tree? YES NO
   2. a full tree? YES NO
   3. a complete tree? YES NO



1. For the tree shown here, answer each of these questions by circling the correct answer.
   1. Is it a two-tree? Yes No



* 1. Is it full? Yes No
  2. Is it complete? Yes No
  3. Is it a binary search tree? Yes No
  4. Is it a heap? Yes No

1. For the tree shown here, answer each of these questions by circling the correct answer.
   1. Is it a two-tree? Yes No



* 1. Is it full? Yes No
  2. Is it complete? Yes No
  3. Is it a binary search tree? Yes No
  4. Is it a heap? Yes No

1. Draw the binary search tree that results from inserting these elements into the tree in the order listed here: **9 8 10 90 11 14 12**



1. Consider the following binary search tree:



Draw the resulting tree if we remove 75 from the tree.



1. Consider the following binary search tree:



Draw the tree that results after rotating right around 75.



1. Consider the tree shown below.



Draw the tree that results after rotating left around 20.



1. Remove 25 from the binary search tree shown below.



Answer:



1. Draw the binary search tree that results from inserting these nodes, in the order listed here: **42 60 28 75 70 35 58 10**



1. Remove 40 from the binary search tree shown below.



Answer:



1. Remove 20 from the binary search tree shown below.





1. Rotate left around 32 in the binary search tree shown below.





1. Draw the binary search tree that results from rotating left around 75 in the tree below. Use the algorithm we studied in class.





1. Draw the binary search tree that results from inserting the following elements into the tree, in the order listed here and using the algorithm we studied in class:

**30 20 72 15 25 50 45**



1. Remove 75 from the binary search tree shown below, using the algorithm we studied in class.





1. Show the order in which the nodes are visited in the binary search tree below if we perform a preorder traversal.



**Preorder traversal**

**50 25 75 60 80 77**

1. Answer the following questions for the tree shown below.
   1. Is this a two-tree? Yes



* 1. Is this a complete tree? No
  2. Is this a full tree? No
  3. Is this tree a chain? No
  4. How many leaves does this tree have? 6
  5. What is the height of this tree? 3

1. In a tree, nodes with the same parent are called
   1. ancestors
   2. descendants
   3. siblings
   4. none of the above
2. A \_\_\_\_ is defined as a tree which is either empty or every node, except for leaf nodes, has two branches.
   1. balanced tree
   2. complete tree
   3. full tree
   4. two-tree
3. Which of the following is/are true of trees?
   1. the root node is at level 1
   2. the height of a tree is the maximum level of any node
   3. a node in a tree can have at most two children
   4. all leaves in a tree are on the same level
4. A balanced tree is always complete.
   1. true
   2. false
5. A complete binary tree is also a full tree.
   1. true
   2. false
6. A full tree is always balanced.
   1. true
   2. false
7. A full tree is a complete binary tree.
   1. true
   2. false
8. A full tree has all its leaves on the same level.
   1. true
   2. false

# Heaps

1. Draw the heap that results from inserting these elements into the heap. Elements must be inserted in the order given here, and you must use the algorithm discussed in class.

42 93 19 10 50 99 12

**Draw Heap Here**



1. Consider the heap shown here. Draw the heap that results from removing 15. You must use the algorithm discussed in class.

**Original Heap**



**Result of Removing 15**



1. In a heap, nodes must be comparable.
   1. true
   2. false
2. Draw the heap that results when you insert these elements into the heap, in the order specified here:  **18 22 95 4 12 19 76**

**DRAW TREE HERE**



1. Remove the top element from the following heap.





1. Consider the heap shown below. Draw the resulting heap when 2 is added to the heap.





1. Consider the heap shown below. Draw the resulting heap when the top element is removed from the heap.





1. Draw the heap that results from inserting these nodes, in the order listed here:

**12 10 8 6 4 2**



1. Remove the top element from the heap shown below.





1. Draw the heap that results from inserting the following elements into a heap, in the order listed here and using the algorithm we studied in class: **25 14 7 3**



1. Show what the heap below will look like if we remove the top element. You must use the algorithm we studied in class.





1. The root node in a heap is called the \_\_\_\_\_.
   1. first element
   2. parent
   3. top
   4. last
2. Which of the following is/are true of heaps?
   1. nodes must be comparable
   2. except for leaves, every node has two children
   3. the value stored in the root of the tree must be the smallest value in the tree
   4. heaps are always balanced