1. **Define the following in terms of Objects/classes in Java.**

* **accessor method**: method used to return the value of a private field/instance variables within a class. Used as part of encapsulation.
* **mutator method**: method used to change the value of a private field/instance variables within a class. Used as part of encapsulation.
* **implicit parameter**: the object reference passed before the name of a method or variable. (ex: this.method();)
* **explicit parameter**: the parameter passed into the parenthesis when a method is called
* **String literal**: sequence of characters represented by enclosed in double quotes
* **scope**: the block of code where a variable can be accessed\mutated
* **null reference**: indicates that the object variable does not refer to any object
* **pass by value**: passing variables into a method passes the literal value of the variable, not its reference (in the case of an object). A new local variable with the same value is created (defined in the parameters), yet any changes to the local literal will not change the passed variable’s value. In Java, objects are only passed by value and not by their references (a changed value within a passed array would still remain after the method is finished).
* **promoting**: Assigning a smaller variable to a larger PDT. For example, when assigning an int to a double, the int is promoted to a double.
* **formal parameter**: the identifier used in a method to represent the actual value that is being passed in a method, once the method is finished the variable would have no value.
* **actual parameter**: the actual value that is passed into the method when a method is called.
* **pass by reference**: Unlike passing by value, passing by reference passes the object’s reference (not the value) to the method. Meaning that by passing by reference a change made to a value in a different scope (the passed method) is preserved when that scope is finished.

1. **Explain the relationship between variable scope and the initialization process.**

In a class a variable can be initialized outside of any method (while still inside the class), meaning that it can be accessed/mutated anywhere within the class. This is a different case for variables that are initialized within a method, as they can only be accessed within that method. Where a variable can be accessed is referred to as its scope. So wherever a variable is initialized within a block of code, it’s scope is bound to that block it was initialized in.

1. **Explain what is happening in RAM in each of these lines of code:**

String m = “My name”;

String r = “Your name”;

r = m;

m = null;

* A new instance of a String is made with the reference variable of m, which point to a location in memory where a sequence of characters are stored with the literal of “My name”.
* Another instance of a String object is made with a reference variable of r, which refers to a different location in memory that stores the literal of “Your name”.
* The reference variable r is now set to point at a copy of the data from reference variable m. Therefore both reference variables reference “My name”, yet in different locations within the memory
* Now the reference variable m is set to null, meaning that it refers to no object

1. **Explain the differences between compareTo(), compareToIgnoreCase(), and equals().**

**equals():** compares two string literals based on their data, but unlike the compareTo methods it only returens true or false based off of whether that data is the literals have the same value.

**compareTo():** Compares two strings lexicographically.Both strings are converted to ASCII values for each of their characters to be compared. If both the strings are equal then this method returns 0. If they are not the same, It returns a positive integer if the first string is lexicographically greater than the second string or the other way around where it would return a negative (based off of the ASCII value).

**compareToIgnoreCase():** does the exact same thing as compareTo(), comparing two strings lexicographically based off of their ASCII values, yet it ignores the case. So if the strings were “HeLLo” and “Hello”, compareToIgnoreCase() would return a 0, where compareTo() would return a negative number.

1. **The Object class - the following methods are required for the AP exam. Define each method in your own words and give your own code example for each method.**

* **toString():**  The default toString() method in Object prints the hash code of the class. By overriding the toString method it can be used to print a proper output. For example toString() is overridden to return a set of data to the reader that is readable .

|  |
| --- |
| Ex:  //dataNumber1 and dataNumber2 were given values when instantiated  public String toString() {  return “classData” + dataNumber1 + “ and “ + dataNumber2;  } |

int n1 = 122;

int n2 = 2;

String result = n1.toString() + n2.toString();

System.out.println(result );

Output: 1222

* **equals() :** The equals method within the Object compares the data between two string literals and returns a boolean. It can be overridden in a different class to check whether two objects have the same data or not. A basic equals() method could just be determining if two objects are the same (same values) such as the example below (this is very simplified, choosing to compare if the passed object is actually an instance of the class and whether their attributes are the same is a more detailed overriden equals() method)

|  |
| --- |
| Ex:  public boolean equals(Object abc) {  return (abc == this);  } |

String s1 = “nono”;

String s2 = “nono”;

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

Output : true

* **hashCode():** The hasCode method returns the hasCode (the memory address of the object as an integer) value (or hash) for each object based on its data. An object will have the same hash. Based off of .equals() objects that are equal have the same hash.

|  |
| --- |
| Ex:  //A objID was instantiated when the object was made  public int hashCode() {  return objID;  } |

String s1 = “hashes”;

String s2 = “hashes”;

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

Output: true

1. **Explain class downcasting. List the pros and cons of downcasting as well as any cautions.**

Downcasting is when a superclass is casted into a subclass, as in SuperClass n = new SubClass(); allowing it to access subclass attributes and behaviours without redundancy.

**Pros**

It can be used to access specific methods or behaviors of a subclass

Reduces the need for redundant code (can just use the superclasses’) and makes it more legible

**Cons and Cautions**

Can not access overridden superclass behaviours within the subclass

If the method does not exist in the superclass an error will occur (unless casted to subclass)

1. **Explain class casting. List the pros and cons of casting as well as any cautions.**

Class casting is when a superclass is casted into a subclass, by (SubClass)super. Allowing you to access subclass methods (unique or overridden) that are not contained in the superclass

**Pros**

Reduces redundancy within the superclass (also makes it more legible)

It can be used to access specific behaviors of a subclass not found in the superclass

**Cons and Cautions**

Can not access overridden superclass methods

Can cause ClassCastException if not careful (more than one subclass inheriting superclass and superclass is downcasting a different subclass) and if neither class has that method

Can look confusing out of context

Can only cast to a specific subclass of that object cant cast to another subclass of the same superclass

1. **Explain PDT casting. List the pros and cons of PTD casting as well as any cautions.**

Primitive Data Type casting is when primitive data types are converted to other PDTs. Although Booleans cannot be casted

**Pros**

Allows operations to occur between two different data types ex. (int) Double + int

Allows for (still somewhat) accurate integer division

Represent values in different data types

**Cons and Cautions**

Accuracy is lost when converting from a data type with more bits (double) to a data type with less bits (int)

long -> double or int -> float may cause loss in accuracy as the transition from a 32 bit to a 64 bit data type can produce “extra precision”

1. **Explain the difference between static/class and instance/member methods.**

Static/class methods do not require an object to use it or to be associated with an instance of a class. It can be run directly by using the class name. (It is associated with the class not the object) Instance/member methods require an instance of an object to be associated with. The object reference is used to call the instance methods of an object. (associated with the class and an object)