Home

UNIT 1 - Primitive Data types

Data types can be either primitive or reference

- Doubles handle much more prescice numbers
  - Booleans have two values
  - Ints have max num smaller than doubles only have specific num of bits
  - When variable is delaired final, its value cannot be changed

Casting

```java
(int)varibleName
(double)variableName
//force implimentation
//This does not round, it truncates it
1.5(int)
//would be 1
```

Arithmatic: only used with int and double

- / * % -
  - if run operation an int, you get an int
  - if run operaiton on double , you get double
  - if an int is manipulated by double it is double
  - Modulus is remainder operaiton takes remainer of division
  - Shorthand
    - Compound assignmnet operator
    -

```
+=
-=
/=
%=
++
```

UNIT 2 - Objects

Class Instantiation

- Instantiation is the process of creating an instance of a class.
- Every object is created using the "new" keyword.
- A constructor is a method with the same name as the class, used to initialize objects.
- Constructors are overloaded when there are multiple constructors with the same name but different parameters.
- "null" is a special value that indicates the absence of reference for data.
- Parameters are usually specified in the header of the constructor.

Types of Methods

- The behavior of objects is defined by methods.
- The signature is the primary way of determining what a method does.
- The following are some types of methods:

- Public
- Static
- Void
- Int
- Double
- Boolean
- String

Using Methods

- Methods are not usually sequential and are only executed when called.
- To call a void method, you should not call it in a place where data would be used if it returned something.
- To call a static method, you can call it without making an object for the class.
- To call a non-static method, you should call it after making an object for the class.

Strings and String Methods

- The String method String(String str) is the best way to make a string.

- String methods include:
  - int length()
  - String substring(int from, int to)
    between 0.0 and 1.0

- `String substring(int from)`: Returns a new string that is a substring of this string, beginning at the specified index and extending to the end of the string.

- `int indexOf(String str)`: Returns the index within this string of the first occurrence of the specified substring.

- `boolean equals(String other)`: Compares this string to the specified object. Note: do not use `==` instead use this.

- `int compareTo(String other)`: Compares two strings lexicographically. Compares two strings ascii code.

Wrapper Classes

- Create and call `Integer` and `Double` methods:
  - `Integer(int value)`: Constructs a newly allocated Integer object that represents the specified int value.
  - `Integer.MIN_VALUE`: Returns the minimum value an Integer can have.
  - Does not change value, just makes it an object.

Math Class

- Only has static methods:
  - `int abs(int x)`: Returns the absolute value of an int value.
  - `double abs(double x)`: Returns the absolute value of a double value.
  - `double pow(double base, double exponent)`: Returns the value of the first argument raised to the power of the second argument.
  - `double sqrt(double x)`: Returns the correctly rounded positive square root of a double value.
  - `double random()`: Returns a pseudo-random double between 0.0 and 1.0.

UNIT 3 - Boolean expressions

When working with evaluative boolean expressions with primitive values, you can test using either:

- == or != - The == operator checks to see if the references are the same.

- The default .equals() method checks to see if the objects are logically equal.
- Relational operators such as <, >, <=, or >=

Conditional statements include:

- If statements - a conditional statement that interrupts the sequential flow
- If else statements - a conditional statement that tests, then has a base case if it is false
- Else if statements - tests a second or more cases after the original if statement

Here is an example of conditional statements:

```
if(condition){
    cool code
}
elseif(condition){
    cool code
}
else{
    cool code
}
```

Logical operators `&&` and `||` result in the following output:

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

When comparing object references using booleans:

- Reference value can be compared with `==` or `!=`.
- Classes normally have their own .equals method or the default.

Differences

- - The == operator checks to see if the references are the same.
- The default .equals() method checks to see if the objects are logically equal.

UNIT 4 - Iteration

For / While loops

While

```
while(bool == true){
//cool code stuff
}

int i = 1;
while(i > 10){
//cool code stuff
i++
}
```

- Set of code that repeats certain number of times
- Boolean expression is evaluated before each iteration of the loop body, including the first
  - Infinite loop when the boolean will never be false
  - When there is a return statement within the loop, when it reaches it, the loop will be done

For

```
for(int i = 0; i < lengthOfSum.length(); i++){
//cool code stuff
}
```

- In for loop initalization statement is only executed once before the first boolean evaluation
- In each iteration of a for loop the increment statement is executed after the entire loop body is executed and before the bolean statement
- Many standard algorithms for string traversals

Nested iteration

```
for(int i = 0; i > 10; i++){
        for(int i = 0; i > 20; i++){
        //cool code stuff
        }
}
```

Very popular for 2d arrays
will repeat statement inside of first loop until it is finished, then go again

UNIT 5 - Classes

Details

- Always public
  - though things inside that class can be private.
- Access to attributes should be kept internal to the class. Therefore, instance variables are designated as private.
- Constructors are designated as public
- Data encapsulation is a technique in which the implementation details of a class are kept hidden from the user
- When designing a class, programmers make decisions about what data to make accessible and modifiable from an external class. Data can be either accessible or modifiable, both, or neither.
- INstance variables are encapsulated by using the private access modifier.
- The provided accessor and mutator methods in a class allow client code and modify datai

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

CONSTRUCTORS:

- An object's state refers to its attributes and values at a given time ind is defined by instance variables belonging to the object. This creates a "has-a" relationship between the object and instance variables.
- Constructors are used to set the initial state of an object, which should include vital initial values for all instance variables
- Constructor parameters are local variables to the constructor and provide data to initialize instance variables

ACESSOR METHODS:

- An accessor method allows other objects to obtain the value of instance variables or static variables
- a non-void method returns a single value. Its header include the return type in place of the keyword void

```
public class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }
}
```

toString METHOD:

- The toString method is an overridden method that is included in the class to provide a description of a specific object. It generally includes what values are stored in the instance data of the objects.

```
public class Car {
    private String make;
    private String model;

    @Override
    public String toString() {
        return make + " " + model;
    }
}
```

MUTATOR METHODS:

- A void method that does not return a value, but changes values of instance or static variables

```
public class Rectangle {
    private int length;
    private int width;

    public void setLength(int length) {
        this.length = length;
    }
}
```

METHODS:

- Methods can only access the private data and methods of a parameter that is a reference to an object when the parameter is the same type as the method's closing class
- Non-void methods with parameters review values through parameters, use those values, and return a computed value of the specified type.
- It is good programming practice to not modify mutable objects that are passed as parameters unless required in the specification
- When an actual parameter is a primitive value, the formal parameter is initialized with a copy of that value. Changes to the formal parameter have no effect on the corresponding actual parameter.

```
public class MathHelper {
    public static int sum(int a, int b) {
        return a + b;
    }
}
```

STATIC METHODS:

- are associated with the class, not objects of the class.
- include the keyword statc in the header before the method name.
- cannot access or changed the values of instance variables.
- an access or change the values of static variables

```
public class Counter {
    private static int count = 0;

    public static void incrementCount() {
        count++;
    }
}
```

UNIT 6 - Arrays

Data collection with similar information

An array allows multiple related items to be represented using a single variable

An array is created using the keyword "new" because it is an object

```
int arr[] = new int[10];
```

Valid index values are 0 - one less than number of elements in array
Anything else will get you out of bounds errors

Traversing the array

Iteration statements can be used to access all of the elements in a array. This is called travering the array.

```
int[] numbers = {2, 4, 6, 8, 10};

for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Enhanced for loops

```
int[] numbers = {2, 4, 6, 8, 10};

for (int number : numbers) {
    System.out.println(number);
}
```

- Header includes a variable (i)
- for each iteration, the enhanced for loop is assigned a copy of an element without using its index
- asiging a new value to the enhanced for loop variable does not change the value in the array

Sorting

- Standard algorithms utilize array traversals to
  - Determie minimum or max vallue
  - compute sum average or mode
  - ext...
  - Shift or rotate elements
  - determine number of elements meeting criteria
  - ext...
- **Selecion sort:**

```
public static void selectionSort(int[] arr) {
    // Loop through the array from the beginning to the second-to-last element
    for (int i = 0; i < arr.length - 1; i++) {
        // Assume the current index is the minimum
        int minIdx = i;
        // Loop through the unsorted part of the array to find the true minimum
        for (int j = i + 1; j < arr.length; j++) {
            // If the current element is smaller than the current minimum, update the minimum
index
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        // Swap the current element with the minimum element
        int temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}
```

UNIT 7 - ArrayLists

General info

- An arraylist is completly mutable and can only contain object refrences
- The arraylist constructor creates an empty list
- When the arraylist is specified you put in the data type

```
 ArrayList<String> myArrayList = new ArrayList<String>();
```

Methods

- int size()
  - Returns number of elements in list
- boolean add(E obj)
  - Appends object to end of list, returns true
- void add(int index, e object)
  - inserts obj at position index moving elements to the right
- E get (int index)
  - Returns element at position index in the list
- E set (int index, E obj)
  - Replaces element at index
- E remove(int index)
  - removes element from index, moving everything to right

```
    // Adding elements to the ArrayList
    myArrayList.add("apple");
    myArrayList.add("banana");
    myArrayList.add("orange");

    // Accessing elements in the ArrayList
    System.out.println("First element: " + myArrayList.get(0));
    System.out.println("Second element: " + myArrayList.get(1));
    System.out.println("Third element: " + myArrayList.get(2));

// Iterating over the ArrayList using a for loop
    System.out.println("All elements:");
```

```
    for (int i = 0; i < myArrayList.size(); i++) {
      System.out.println(myArrayList.get(i));
    }

    // Removing an element from the ArrayList
    myArrayList.remove(1);

    // Checking the size of the ArrayList
    System.out.println("Size of ArrayList after removing an element: " + myArrayList.size());

    // Clearing the ArrayList
    myArrayList.clear();

    // Checking the size of the ArrayList after clearing it
    System.out.println("Size of ArrayList after clearing it: " + myArrayList.size());
  }
}
```

Traversing

- Iteration statements can be used to access all elements and traverse array list
- Deleting elements during a traversal requirs using special tequniques to avoid skipping elements
- The indicies for an ArrayList start at 0 and end at the .Size() - 1 accesing index value out of the range will result in an out of bounds exeption

Searching / sorting

- Standard algorithms for searching arraylists
  - Sequential
- Seleciton sort and insertion sort are iterative sorting algorithms that could be used to sort elements in an ArrayList

```
    ArrayList<Integer> myList = new ArrayList<>(Arrays.asList(1, 3, 5, 7, 9));

    int index = Collections.binarySearch(myList, 5);

    if (index >= 0) {
      System.out.println("Element found at index: " + index);
    } else {
      System.out.println("Element not found");
    }
  }
```

UNIT 8 - 2D arrays

General info:

- 2D arrays are stored as arrays of arrays. Therefore, the way 2D arrays are created and indexed is similar to 1D array objects.

```
int[][] = new int [3][6];
```

Traversing an array

- nested iteration statements are used to traverse and access 2d elements

```
int[][] matrix = {{1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9}};

// Traverse the matrix row by row
```

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[0].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

- When applying search algorithms to 2d arrays, each row mush be acessed then have the sequential or linear search applied to each row

```
    public static int[] linearSearch(int[][] matrix, int target) {
        int numRows = matrix.length;
        int numCols = matrix[0].length;

        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                if (matrix[i][j] == target) {
                    int[] result = {i, j};
                    return result;
                }
            }
        }

        return null; // target not found
    }
```

UNIT 9 - Inheritance

Class hierarchy

- developed by having common attributes and behaviors of the related classes in a single class by a superclass
- Classes that extend a superclass, are called sublasses and can drap upon the existing attributes and behaviors
-

```
class Vehicle {
    private String brand;
    private int year;

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public String getBrand() {
        return brand;
    }

    public int getYear() {
        return year;
    }

    public void accelerate() {
        System.out.println("Vehicle is accelerating");
    }
}

// This is the subclass
class Car extends Vehicle {
```

```
    private int numDoors;

    public Car(String brand, int year, int numDoors) {
        super(brand, year);
        this.numDoors = numDoors;
    }

    public int getNumDoors() {
        return numDoors;
    }

    public void honk() {
        System.out.println("Car is honking");
    }
}

// This is the main method that creates objects of the classes and calls their methods
public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle("Toyota", 2021);
        Car car = new Car("Honda", 2022, 4);

        System.out.println(vehicle.getBrand()); // Toyota
        System.out.println(car.getBrand()); // Honda
        System.out.println(car.getNumDoors()); // 4

        vehicle.accelerate(); // Vehicle is accelerating
        car.accelerate(); // Vehicle is accelerating
        car.honk(); // Car is honking
    }
}
```

Method overriding

- Multiple classes contain comon attributes and behaavoirs
- Proogrammerrs create new classes containing the behavoirs forming a heirarchy
- Method overriding occurs when a public method in a subclass has the same method signitre as a public method in the superclass

```
// This is the superclass
class Animal {
    public void speak() {
        System.out.println("An animal speaks");
    }
}

// This is the subclass
class Dog extends Animal {
    public void speak() {
        System.out.println("A dog barks");
    }
}

// This is the main method
public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();
```

```
        animal.speak(); // An animal speaks
        dog.speak(); // A dog barks
    }
}
```

"Super" keyword

- used to call a superclasses constructors and methods
- superclass methdod is called imn a subclass by using the keyword super followed by the method name and paassing the appropriate parameters

```
// This is the superclass
class Animal {
    public Animal() {
        System.out.println("An animal is created");
    }
}

// This is the subclass
class Dog extends Animal {
    public Dog() {
        super(); // call the constructor of the superclass
        System.out.println("A dog is created");
    }
}

// This is the main method
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
    }
}
```

Polymorphism

definition - it can change into any different implimentation

```
// This is the superclass
class Animal {
    public void makeSound() {
        System.out.println("An animal makes a sound");
    }
}

// This is the subclass
class Dog extends Animal {
    public void makeSound() {
        System.out.println("A dog barks");
    }
}

// This is another subclass
class Cat extends Animal {
    public void makeSound() {
        System.out.println("A cat meows");
    }
}

// This is the main method
public class Main {
```

```
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.makeSound(); // A dog barks
        animal2.makeSound(); // A cat meows
    }
}
```

In the `Main` class, we create objects of the `Dog` and `Cat` classes and assign them to variables of type `Animal`. This is possible because both `Dog` and `Cat` are subclasses of `Animal`. We then call the `makeSound()` method on both objects. Since both `Dog` and `Cat` override the `makeSound()` method of the `Animal` class, the actual behavior that we see when we call `makeSound()` depends on the specific subclass instance that we're calling it on. This is an example of polymorphism, where different objects of the same class hierarchy can have different behavior.

Java COSMIC Superclass

Object class

- the superclass to everything in java
- comes with a few methods (can be customized)

```
boolean equals(Object other);
String toString();
```

UNIT 10 - Recursion

General info

- Method that calls itself
- Contains atleast one base case that halts the recursion
- For every recursive method, there is an equivilant interation solution to the problem
  `note: writing recursive methods are out of the scope of the AP exam`

Recursive searching and sorting

- Bianary search starts at the middle of the sorted array and eliminates half of the array until the desired value is found

```
public class BinarySearch {

    public static int binarySearch(int[] arr, int target, int left, int right) {
        if (left > right) {
            return -1; // target not found
        }

        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid; // target found at index mid
        } else if (arr[mid] > target) {
            return binarySearch(arr, target, left, mid - 1); // search in left half
        } else {
            return binarySearch(arr, target, mid + 1, right); // search in right half
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int target = 7;
        int index = binarySearch(arr, target, 0, arr.length - 1);
        if (index != -1) {
```

```
            System.out.println("Target found at index " + index);
        } else {
            System.out.println("Target not found");
        }
    }
}
```

- Merge sort is a recursive sorting algorithm that works by dividing an array into two halves, sorting each half separately, and then merging the two sorted halves back together

```
public class MergeSort {

    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid); // sort left half
            mergeSort(arr, mid + 1, right); // sort right half
            merge(arr, left, mid, right); // merge two halves
        }
    }


    public static void merge(int[] arr, int left, int mid, int right) {
        // create temporary arrays to hold the left and right halves
        int[] leftArr = new int[mid - left + 1];
        int[] rightArr = new int[right - mid];

        // copy elements from arr to the temporary arrays
        for (int i = 0; i < leftArr.length; i++) {
            leftArr[i] = arr[left + i];
        }
        for (int i = 0; i < rightArr.length; i++) {
            rightArr[i] = arr[mid + 1 + i];
        }

        // merge the two halves back into arr
        int i = 0;
        int j = 0;
        int k = left;
        while (i < leftArr.length && j < rightArr.length) {
            if (leftArr[i] <= rightArr[j]) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        // copy any remaining elements from leftArr and rightArr to arr
        while (i < leftArr.length) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }
        while (j < rightArr.length) {
            arr[k] = rightArr[j];
            j++;
            k++;
```

```
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 4, 7, 1, 3, 2, 6};
        mergeSort(arr, 0, arr.length - 1);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```