Native Data Types, Static, Wrapper Classes, Arrays and Strings, valid identifiers

Native Data Types

1. In Java, native (or *primitive*) data types are the most basic data types that are directly supported by the language. They are not objects and are therefore stored in a way that is more memory efficient. Java has 8 primitive data types, which can be categorized into four main groups:

Integer Types

- **byte**: 8-bit signed integer, range: -128 to 127
- **short**: 16-bit signed integer, range: -32,768 to 32,767
- **int**: 32-bit signed integer, range: -2^{31} to $2^{31} - 1$
- **long**: 64-bit signed integer, range: -2^{63} to $2^{63} - 1$

Floating-Point Types

- float: 32-bit IEEE 754 floating-point, precision up to 7 decimal digits
- double: 64-bit IEEE 754 floating-point, precision up to 15 decimal digits

Character Type

- **char**: 16-bit Unicode character

Boolean Type

1. boolean: Represents a value of either true or false

Native Data Types

Important information's about Native Data Types in Java:

1. In Java, String is not a native (or primitive) data type. Instead, it is a class in the java.lang package, which makes it a reference type.
2. Reference types(objects made from classes) can be set to null, meaning they don’t point to any object. Trying to access a method or field on null results in a NullPointerException, which is a common beginner error. Native/Primitive data types cannot be null and instead have default values (e.g., 0 for int, false for boolean).

Static

1. You cannot use instance variables(properties) from a static method
2. Default values for static variables:

```
class MyClass {
    static int staticInt;
    static boolean staticBoolean;
    static char staticChar;
}

// Values of static variables without initialization
System.out.println(MyClass.staticInt);       // Output: 0
System.out.println(MyClass.staticBoolean);   // Output: false
System.out.println(MyClass.staticChar);      // Output: '\u0000'
```

QUIZ

What is the use of the static keyword in Java?

A) It indicates that a variable or method belongs to the class, rather than instances of the class.

B) B) It signifies that a variable or method can only be accessed within the same package.

C) C) It denotes that a variable or method can only be accessed by subclasses.

D) D) It specifies that a variable or method is transient and can be changed dynamically during runtime.

QUIZ ANSWER

Fill in the blank to make the code compile:

A) It indicates that a variable or method belongs to the class, rather than instances of the class.

B) B) It signifies that a variable or method can only be accessed within the same package.

C) C) It denotes that a variable or method can only be accessed by subclasses.

D) D) It specifies that a variable or method is transient and can be changed dynamically during runtime.

Correct Answer: A) It indicates that a variable or method belongs to the class, rather than instances of

QUIZ

Fill in the blank to make the code compile:

A. cat.name
B. cat-name
C. cat.setName
D. cat[name]

```
package animal;
public class Cat{
  public String name ;
  public static void main (String[] meow){
    Cat cat = new Cat();
    ______= "Sadie";
  }
}
```

QUIZ ANSWER

Fill in the blank to make the code compile:

A. cat.name

B. cat-name

C. cat.setName

D. cat[name]

A. Java uses dot notation to reference instance variables in a class, making Option A correct.

QUIZ

Which statement is true about primitives?

A. Primitive types begin with a lowercase letter.

B. Primitive types can be set to null.

C. String is a primitive.

D. You can create your own primitive types.

QUIZ ANSWER

Which statement is true about primitives?

A. Primitive types begin with a lowercase letter.

B. Primitive types can be set to null.

C. String is a primitive.

D. You can create your own primitive types.

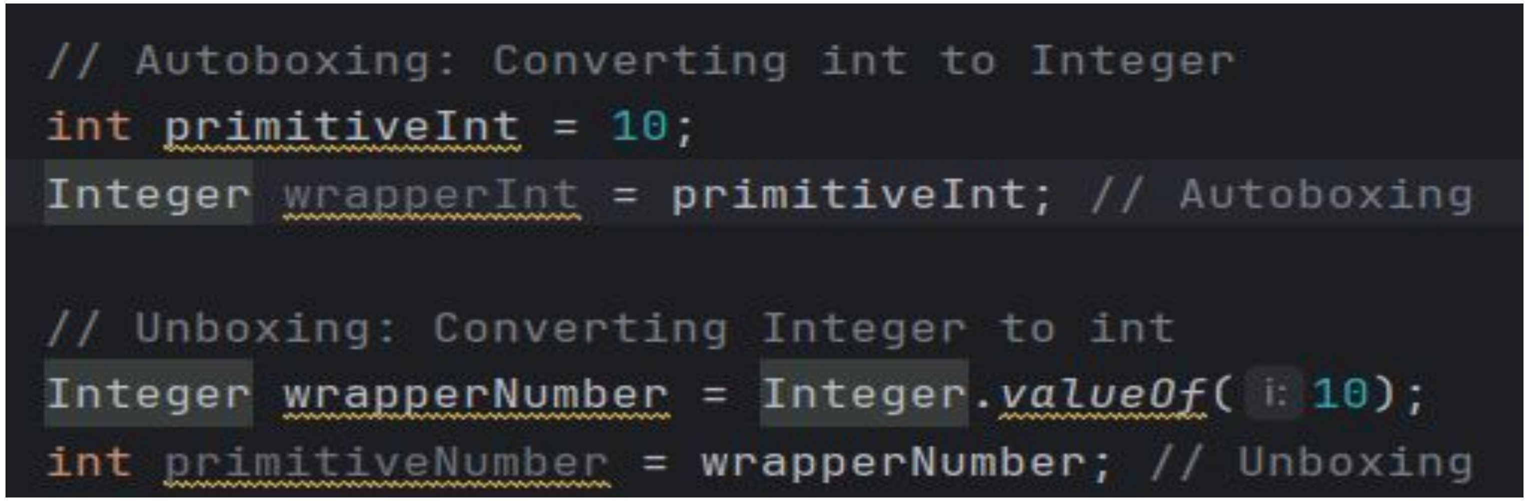
A. An example of a primitive type is int. All the primitive types are lowercase, making Option A correct. Unlike object reference variables, primitives cannot reference null. String is not a primitive as evidenced by the uppercase letter in the name and the fact that we can call methods on it. You can create your own classes, but not primitives.

Wrapper Classes

1. In Java, primitive data types (e.g., int, double, boolean) have corresponding wrapper classes (e.g., Integer, Double, Boolean) that represent them as objects.
2. Autoboxing allows you to assign primitive values directly to wrapper class objects, and Java automatically converts the primitive value into the corresponding wrapper object. Similarly, when a wrapper object is used in a context that expects a primitive type, Java automatically unboxes the object to retrieve its primitive value.

3. These classes provide utility methods and allow null values for primitive data types.
   1. Integer num = Integer.valueOf(5);// Wrapper class for int
   2. int value = num.intValue(); // Extracting int value from Integer object

Creating objects of the wrapper classes



```java
// Autoboxing: Converting int to Integer
int primitiveInt = 10;
Integer wrapperInt = primitiveInt; // Autoboxing

// Unboxing: Converting Integer to int
Integer wrapperNumber = Integer.valueOf( i: 10);
int primitiveNumber = wrapperNumber; // Unboxing
```

Autoboxing allows you to assign primitive values directly to wrapper class objects, and Java automatically converts the primitive value into the corresponding wrapper object. Similarly, when a wrapper object is used in a context that expects a primitive type, Java automatically unboxes the object to retrieve its primitive value.

1. You can create objects of all the wrapper classes in multiple ways:
2. Assignment - By assigning a primitive to a wrapper class variable (autoboxing)
3. Static methods - By calling static method of wrapper classes, like, `valueOf()`

Assignment

```
1.  Boolean bool1 = true;
2.  Character char1 = 'a';
3.  Byte byte1 = 10;
4.  Double double1 = 10.98;
```

Using static method valueOf()

```
1.  Boolean bool4 = Boolean.valueOf(true);
2.  Boolean bool5 = Boolean.valueOf(true);
3.  Boolean bool6 = Boolean.valueOf("TrUE");
4.  Double double4 = Double.valueOf(10);
```

WrapperClassExample

```
public class WrapperClassExample {
    public static void main(String[] args) {
        // Conversion
        int primitiveInt = 10;
        Integer wrapperInt = Integer.valueOf(primitiveInt); // Convert int to
Integer
        int backToInt = wrapperInt.intValue(); // Convert Integer to int
        System.out.println("Converted int: " + backToInt);

        double primitiveDouble = 10.5;
        Double wrapperDouble = Double.valueOf(primitiveDouble); // Convert
double to Double
        double backToDouble = wrapperDouble.doubleValue(); // Convert Double to
double
        System.out.println("Converted double: " + backToDouble);
```

Arrays

1. Arrays in Java are used to store multiple values of the same type.
2. They have a fixed size and can store primitive or reference types

```
int[] numbers = new int[5]; // Creating an array of integers with size 5
numbers[0] = 1; // Assigning value to the first element
```

Arrays

1. To declare an array in Java, you specify the type of elements the array will hold, followed by square brackets [], and then the array name

```
int[] numbers; // Declares an integer array named numbers
```

Arrays

1. Arrays can be initialised using an array initializer or by specifying the size of the array and then assigning values to each element.

```
int[] numbers = {1, 2, 3, 4, 5}; // Initializes an array with values

int[] numbers = new int[5]; // Initializes an array with size 5
```

Arrays

1. You can access elements of an array using the index of the element. Array indices start from 0.

```
int[] numbers = {1, 2, 3, 4, 5};

int firstElement = numbers[0]; // Accesses the first element (1)
```

Arrays

1. The length of an array (i.e., the number of elements it can hold) can be obtained using the length property.

```
int[] numbers = {1, 2, 3, 4, 5};

int arrayLength = numbers.length; // arrayLength is 5
```

Arrays

1. You can use loops (such as for loops) to iterate over the elements of an array

```
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Arrays

1. Java supports multidimensional arrays, which are arrays of arrays. For example:

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int element = matrix[1][2]; // Accesses the element in the second row and third column (6)
```

Arrays

1. In Java, you can also create arrays of objects. For example, an array of String objects:

```
String[] names = {"Alice", "Bob", "Charlie"};
```

Arrays

```
public class ArraysDemo {
    public static void main(String[] args) {
        // Declaration and Initialization
        int[] numbers = {1, 2, 3, 4, 5}; // Initializes an array
with values
        int[] numbers2 = new int[5]; // Initializes an array with
size 5

        // Accessing Elements
        int firstElement = numbers[0]; // Accesses the first element
(1)

        // Array Length
        int arrayLength = numbers.length; // arrayLength is 5
```

Assigning values to arrays indexes

You can assign values to array indexes in this way:

```
int[] myArray = new int[5];

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

            myArray[i] = i * 10; // Assigning a value to the current index

  }
```

String

1. String is a class that represents a sequence of characters.
2. It is widely used to store and manipulate text.
3. You can create a String object using a string literal or by using the new keyword with the String constructor.

```
String str1 = "Hello"; // Using string literal

String str2 = new String("World"); // Using the String constructor
```

String

1. String objects in Java are immutable, which means once a String object is created, its value cannot be changed. If you want to modify a String, a new String object is created.

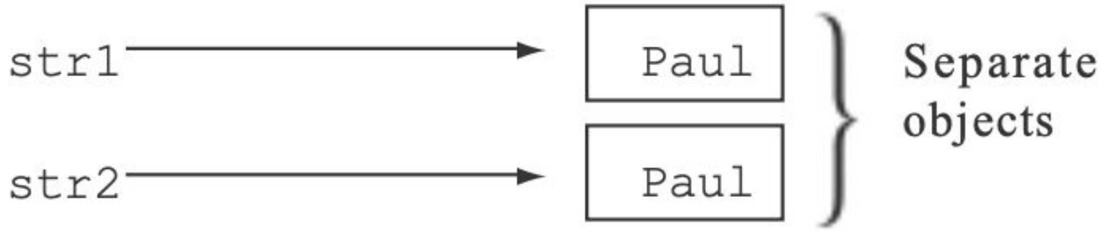
```
String str = “Hello"; // String Literal

str = str + " World"; // This creates a new String object
```

String objects and literal

1. String objects created using the operator new always refer to separate objects, even if they store the same sequence of characters.
2. A comparison of the String reference variables **`str1`** and **`str2`** prints false.
3. The operator == compares the addresses of the objects referred to by the variables **`str1`** and **`str2`**.
4. Even though these String objects store the same sequence of characters, they refer to separate objects stored at separate locations.

```
String str1 = new String (“Paul”);
String str2 = new String (“Paul”);
System.out.println(str1 == str2);
```



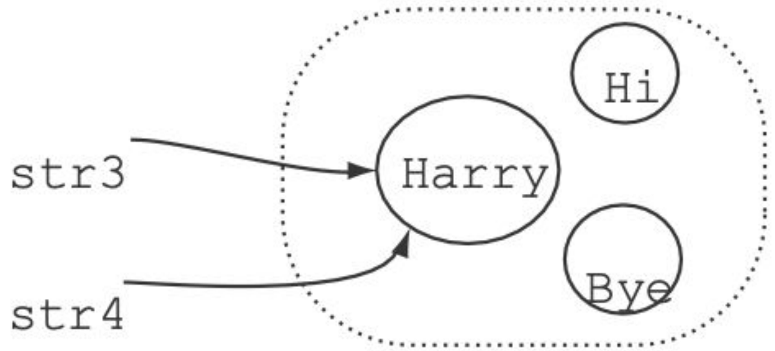
String objects and literal

1. String objects created using the operator new always refer to separate objects, even if they store the same sequence of characters.
2. In the case of variables str3 and str4, the objects are created and stored in a pool of String objects.
3. Before creating a new object in the pool, Java searches for an object with similar contents.
4. When the following line of code executes, no String object with the value "Harry" is found in the pool of String objects:

```
String str3 = “Harry”;

String str4 = “Harry”;

System.out.println(str3 == str4);
```



This is a feature from Java to optimize

String object and literal

Memory allocation

1. String literals are stored in the string pool, a special area of the Java heap memory.
2. When a string literal is encountered, the JVM checks the string pool to see if an identical string already exists.
3. If it does, the existing string is reused; otherwise, a new string is added to the pool.
4. String objects created with the new keyword are stored in the regular heap memory, and a new object is created every time, regardless of whether an identical string already exists.

String object and literal

Comparisons

1. When comparing string literals with the == operator, Java compares their references(memory address), not their contents.
2. This means that two string literals that contain the same value will be equal (== will return true) if they refer to the same object in the string pool.
3. When comparing String objects with ==, Java again compares their references.
4. To compare the contents of two strings, you should use the equals() method, which is overridden in the String class to compare the actual contents of the strings.

String object and literal

Performance

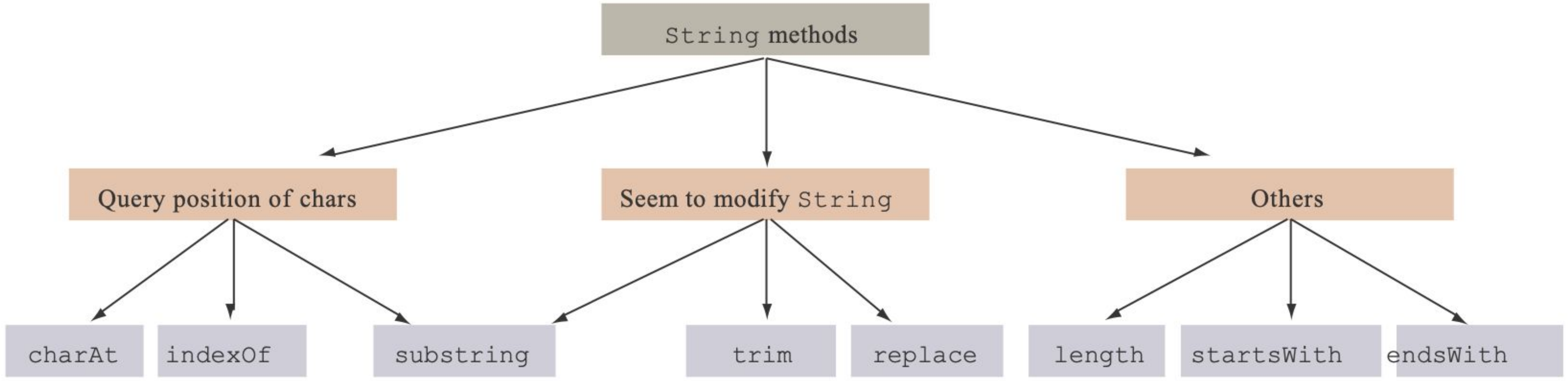
1. String literals are generally more efficient than String objects created with new.
2. Since string literals are stored in the string pool and reused when possible, they can reduce memory usage and improve performance compared to creating new String objects.

String

1. You can concatenate strings using the + operator or the `concat()` method. For example:

```
String str1 = "Hello";
String str2 = "World";
String greeting = str1 + " " + str2; // Using the + operator
String message = str1.concat(" ").concat(str2); // Using the concat() method
```

Methods of the class String



String

1. You can get the length of a String using the **`length()`** method.

```
String str = "Hello";
int length = str.length(); // length is 5
```

String

1. You can call string methods in this way

“Hello”.charAt(3)

1. You can access individual characters in a String using the **`charAt()`** method. The index of the first character is 0.String str = "Hello";

String str = "Hello";

char firstChar = str.charAt(0); // firstChar is 'H’

String

1. You can compare String objects using the **`equals()`** method for content comparison and == operator for reference comparison.

```
String str1 = "Hello";

String str2 = "Hello";

boolean isEqual = str1.equals(str2); // true

boolean isSameObject = str1 == str2; // true (since both point to
the same object in the string pool)
```

String

1. You can format String objects using the `format()` method of the String class or using printf() method of the PrintStream class

String formattedString = String.format("The value of pi is %.2f", Math.PI); // "The value of pi is 3.14"

System.out.printf("The value of pi is %.2f", Math.PI); // Prints "The value of pi is 3.14"

String

```
public class StringDemo {
    public static void main(String[] args) {
        // Creation
        String str1 = "Hello"; // Using string literal
        String str2 = new String("World"); // Using the String constructor

        // Immutability
        String str = "Hello";
        str = str + " World"; // Creating a new String object

        // Concatenation
        String str3 = str1 + " " + str2; // Using the + operator
        String str4 = str1.concat(" ").concat(str2); // Using the concat() method
```

Equality in Strings

There are different ways to check if two strings are equals in java:

- equals(Object obj): This method checks if the current string is equal to another object obj. It returns true if the compared object is also a string and has the same characters as the current string, in the same order. It is case-sensitive, meaning it distinguishes between uppercase and lowercase characters.

- equalsIgnoreCase(String anotherString): This method checks if the current string is equal to another string anotherString, ignoring case differences.

QUIZ

Which is correct about an instance variable of type String?

a. It defaults to an empty string.

b. It defaults to null.

c. It does not have a default value.

d. It will not compile without initializing on the declaration line.

QUIZ ANSWER

Which is correct about an instance variable of type String?

a. It defaults to an empty string.

b. It defaults to null.

c. It does not have a default value.

d. It will not compile without initializing on the declaration line.

B. Instance variables have a default value based on the type. For any non-primitive, including String, that type is a reference to null. Therefore Option B is correct. If the variable was a local variable, Option C would be correct.

Valid identifiers(names) for your variables

1. A valid identifier starts with a letter (a–z, upper- or lowercase), a currency sign, or an underscore. There is no limit to its length.
2. A valid identifier can contain digits but not in the starting place.
3. A valid identifier can use the underscore and currency sign at any position of the identifier.
4. A valid identifier can't have the same spelling as a Java keyword, such as switch.
5. A valid identifier can't use any special characters, including !, @, #, %, ^, &,*, (,),', :, ;, [, /, \, and }.