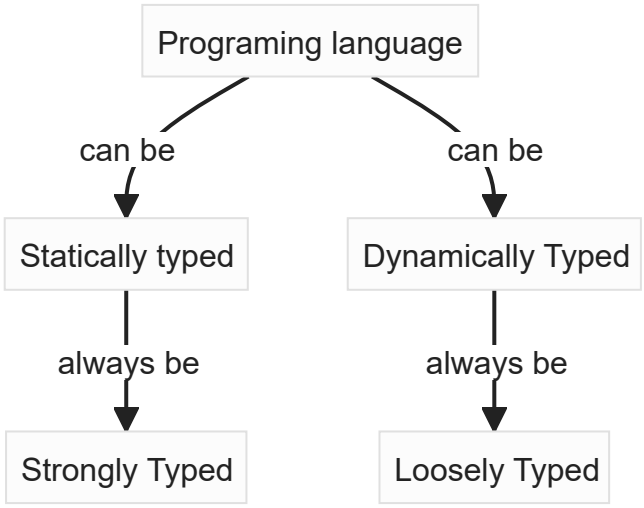
JAVA Core 3

3. Data Types
    1. Primitive data types and storage management
    2. Non-Primitive data types and storage management
    3. Creating custom data types, and Entities in real world

3. Data Types

A data type is a classification that specifies which type of value a variable can hold. It defines the operations that can be done on the data, the meaning of the data, and the way values of that type can be stored.

Programing languages uses various methods for working on the data types, and also implements a type safety mechanism which enhances the error handling for any programming language.



1. **Statically Typed :**
    - In a statically typed language, the type of a variable is known at **compile-time**. This means that you must declare the type of a variable before you can use it.
    - The compiler enforces type checking during the compilation process.
    - Errors related to type mismatches are caught at compile-time.
    - That is why it is easier to write more robust and predictable code.
    - Remember it is not necessary that you need to explicitly write the type of the variable it totally depends on the syntax and the language.
    - For Example : Go is a statically typed language but the syntax of Golang lets it to inference data type from Right Hand Side assignment

```
// Golang : demonstrating syntax and statically typed does not co realte
package main

func main() {

        // normally all other Statically Typed programming language have
        // syntax simillar to this where writting data type is necessary
        var x int64 = 90

        // Golang specific syntax
        // in this type is determined according to the assigned value
```

```
        // but it is assigned at compile time so it is to be said statically
typed
        str := "This is a string"

}
```

2. **Dynamically Typed :**
    - In a dynamically typed language, the type of a variable is determined at **runtime.**
    - You don't need to explicitly declare the type of a variable, it is inferred based on the value assigned to it.
    - The above statement does not corelate by the syntax it is the language that decides as like in Go we can use the same but it checks the type at compile time.
    - Type checking is performed during runtime
    - Errors related to type mismatches are discovered when the code is executed.
3. **Strongly Typed :**
    - In a strongly typed language, the type system is strict, and type safety is enforced.
    - It means that the compiler or interpreter checks types and prevents operations that involve incompatible types.
    - A variable will always be having the value of same data type that it has been given while declaring or defining.
    - Strong typing can help catch errors and prevents un expected behavior of the program.
4. **Loosely Typed :**
    - Loosely typed languages are flexible with how variables are used, allowing for implicit type conversions.
    - A variable can have the value of different data type that it has been given while declaring or defining, implicit type conversions happen as we do so.
    - Operations may be performed on variables of different types without explicit casting.
    - Loosely typed languages prioritize flexibility over strict type enforcement.

3.1 Primitive data types and storage management

In java There are 8 primitive data types whose memory is not allocated in the heap section of the memory, which means the object of these 8 primitive data types is not created and we can use them as like in C/C++, that makes java a partially Object Oriented language. The 8 data types are as follows :

Data Type	Range	Size (in bytes)	Format Specifier	Literal
byte	-128 to 127	1	%d	N/A
short	-32,768 to 32,767	2	%d	N/A
int	-2^31 to 2^31-1	4	%d	N/A
long	-2^63 to 2^63-1	8	%d or %ld	L or l
float	(7 significant digits)	4	%f or %e or %g	F or f
double	(15 significant digits)	8	%f or %e or %g	N/A
char	0 to 65,535 (UTF-8)	2	%c	N/A
boolean	true or false	Not applicable	N/A	true, false

```
public class DataTypes {
    public static void main(String[] args) {

        int decimal = 42;
        int octal = 052;
        int hex = 0x2A;

        double pi = 3.14;
        char letter = 'A';
        boolean flag = true;
        String message = "Hello, World!";
        byte smallNumber = 10;
        short smallInteger = 1000;
        long bigNumber = 123456789L;

                float fNumber = 23.569F;
                double dNumber = 78.909745;

        System.out.println(decimal);
        System.out.println(octal);
        System.out.println(hex);
        System.out.println(pi);
        System.out.println(letter);
        System.out.println(flag);
        System.out.println(message);
        System.out.println(smallNumber);
        System.out.println(smallInteger);
```

```
        System.out.println(bigNumber);
        System.out.println(fNumber);
        System.out.println(dNumber);
    }
}
```

While working with the primitive data, it is necessary that this data can be used seamlessly with other Classes, however these classes works on Objects now the problem arises how can we convert these primitive data types in Object, In here java introduces a new concept known as Wrapper Classes these classes are the wrapper around these primitive data types. That is how we can use the values of primitive data types with the Objects in java.

The Assigned value for the wrapper class is allocated in the Heap while pointing from the Stack, There is a concept of autoboxing and unboxing in java that allow automatic conversion between primitive data types and their corresponding wrapper classes. These features were introduced to simplify the process of working with primitive types and objects, especially for collections framework and Generics.

1. **Auto Boxing :**
    - Autoboxing is the process of automatically converting a primitive data type to its corresponding wrapper class. This happens implicitly when you assign a primitive value to a wrapper class.
2. **UnBoxing :**
    - Unboxing is the process of automatically converting a wrapper class object to its corresponding primitive data type. This also happens implicitly when you assign a wrapper class object to a primitive variable.

```
int primitiveInt = 42;

// Autoboxing: converting int to Integer
Integer wrappedInt = primitiveInt;

Integer wrappedInt2 = 42;

// Unboxing: converting Integer to int
int primitiveInt2 = wrappedInt;
```

Purpose of Wrapper Classes :

- Wrapper classes provide methods to convert primitive data types into objects and vice versa.
- Java Generics do not support primitive data types. By using wrapper classes, you can use generic classes and collections with these types.
- Wrapper classes can represent null values, which is not possible with primitive data types.

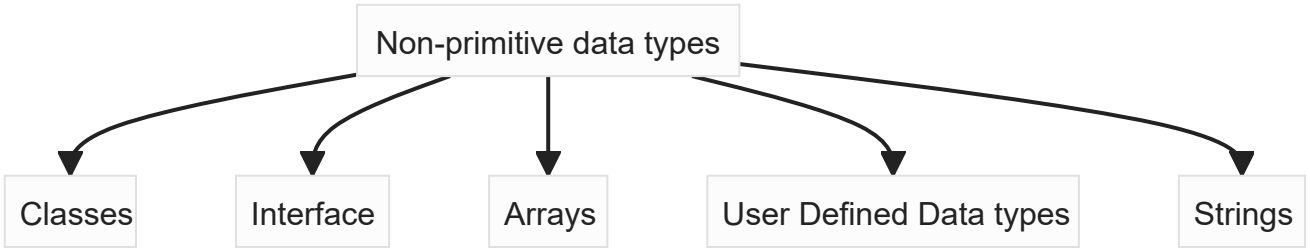
- Wrapper classes provide methods for parsing and converting string representations of numbers.

Primitive Type	Wrapper Class	Example Usage By Wrapper Class	Most Used Methods and Fields
byte	Byte	`Byte b = 42;`	`byteValue()`, `parseByte()`, `MAX_VALUE`
short	Short	`Short sh = 100;`	`shortValue()`, `parseShort()`, `SIZE`
int	Integer	`Integer i = 123;`	`intValue()`, `parseInt()`, `MIN_VALUE`
long	Long	`Long l = 123L;`	`longValue()`, `parseLong()`, `MAX_VALUE`
float	Float	`Float f = 3.14f;`	`floatValue()`, `parseFloat()`, `NaN`
double	Double	`Double d = 3.14;`	`doubleValue()`, `parseDouble()`, `POSITIVE_INFINITY`
char	Character	`Character ch = 'A';`	`charValue()`, `toString()`, `TYPE`
boolean	Boolean	`Boolean bl = true;`	`booleanValue()`, `parseBoolean()`, `TRUE`

In java Constants are stored using all Upper case by convention

3.2 2. Non-Primitive data types and storage management

In Java, non-primitive data types are also known as reference types or objects. These types include classes, interfaces, arrays, and other user-defined types. Unlike primitive data types (int, float, boolean, etc.), which store the actual values, non-primitive data types store references to objects in stack memory and the actual value in heap memory.



1. **Classes:**
   - Classes are user-defined data types in Java.
   - They serve as blueprints for creating objects.
   - Objects are instances of classes, and each object has its own set of data members (fields) and methods.
2. **Interfaces:**
   - Interfaces define a contract for classes to implement.
   - They can include constants and method signatures.
   - Classes implement interfaces, providing concrete implementations for the defined methods.
3. **Arrays:**
   - Arrays are used to store multiple values of the same type.
   - They can be of primitive types or reference types.
   - Arrays are objects in Java.
4. **Strings:**
   - Strings are sequences of characters in Java.
   - The `String` class is part of the Java Standard Library.
   - Strings are immutable, meaning their values cannot be changed after creation.

In Java it is important to understand how the object is referenced in the heap and when does garbage collector works on the un referenced objects.

1. **Object Creation:**
   - Objects are created using the `new` keyword, followed by a constructor call.
   - The `new` keyword allocates memory for the object and invokes the constructor to initialize the object.

     ```
     AnyClass obj = new AnyClass(); // Object creation
     ```

2. **Heap Memory:**
    - Objects in Java are stored in the heap memory.
    - The heap is a region of memory managed by the Java Virtual Machine (JVM).
    - Objects persist in the heap until they are no longer referenced and are eligible for garbage collection.
3. **References:**
    - Non-primitive variables store references to objects, not the actual objects.
    - When an object is created, a reference to that object is returned.
    - Multiple variables can reference the same object.

```
AnyClass obj1 = new AnyClass();
AnyClass obj2 = obj1; // obj2 references the same object as obj1
```

4. **Garbage Collection:**
    - Java uses automatic garbage collection to reclaim memory occupied by objects that are no longer reachable.
    - An object is eligible for garbage collection if there are no references pointing to it.

```
MyClass obj1 = new MyClass();
MyClass obj2 = new MyClass();
obj1 = null; // obj1 no longer references the object
// The object created by obj1 is eligible for garbage collection
```

5. **Null Reference:**
    - A reference variable can be set to `null`, indicating that it does not refer to any object.
    - Setting a reference to `null` makes the object it was referencing eligible for garbage collection.

```
AnyClass obj = new AnyClass();
obj = null; // obj no longer references the object, making it eligible for
garbage collection
```

3.3 Creating custom data types, and Entities in real world

In Java, creating custom data types involves defining classes to model real-world entities, and this process is fundamental to object-oriented programming (OOP). OOP allows you to represent real-world objects as instances of classes, encapsulating data (attributes) and behavior (methods) into cohesive units. Let's go through the process step by step:

1. **Identify the Real-World Entity**
   - Start by identifying the real-world entity you want to model. For example, let's consider a `Person`.
2. **Define the Class**
   - Define a class that represents the identified entity. The class should encapsulate the attributes (data) and behavior (methods) of the entity.
   - In the below example, `Person` is a class with attributes (`name` and `age`), a constructor to initialize these attributes, and a method (`introduce()`) to display information about the person. Getters and setters are optional and provide controlled access to the attributes.

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

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

    // Methods
    public void introduce() {
        System.out.println("Hello, my name is " + name + " and I am " + age + "
years old.");
    }

    // Getters and setters (optional)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

3. **Create Objects**
    - Instantiate objects of the class to represent specific execution flow for the application.

```
public class Main {
    public static void main(String[] args) {
        // Creating instances of the Person class
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);
        // Using methods
        person1.introduce();
        person2.introduce();
        // Modifying attributes using setters
        person1.setAge(31);
        person2.setName("Robert");
        // Updated information
        person1.introduce();
        person2.introduce();
    }
}
```

4. **Use of Constructors and Overloading**
    - Constructors provide a way to initialize object state. Overloading allows a class to have multiple constructors with different parameter lists.

```
public class Person {
    private String name;
    private int age;
    // Default constructor
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }
    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }
    public void introduce() {
        System.out.println("Hello, my name is " + name + " and I am " + age + "
years old.");
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

Practice : create an custom class Slice which gives us all the functionality of a dynamic integer array, having property of length and handles data type range exception, it should automatically increase its size, having method to add, remove and update an element.