### Java Key Features:

1. **Lambda Expressions**: Java supports lambda expressions for functional programming, enabling concise representation of functions and passing behavior as an argument.
2. **Date and Time API (java.time)**: A comprehensive API for handling date and time, including LocalDate, LocalTime, and ZonedDateTime.
3. **Local Variable Type Inference (var)**: Introduced in Java 10, it allows the compiler to infer the type of local variables automatically.
4. **Immutable Collections**: Immutable lists and sets can be created using List.of(...) and Set.of(...), enhancing immutability in Java code.
5. **Switch Expressions**: Java supports switch as an expression, allowing it to return values.
6. **Sealed Classes**: These restrict which classes can extend or implement a class/interface, allowing more control over class hierarchies.
7. **Hidden Classes**: Introduced to allow dynamically generated classes that are not discoverable through classpath scanning.
8. **EdDSA Digital Signature Algorithm**: Java supports the Edwards-curve Digital Signature Algorithm for cryptographic operations.
9. **Records**: A concise syntax for immutable data carriers, reducing boilerplate code for classes that primarily serve as data holders.
10. **Vector API**: Supports vectorized computations, leveraging SIMD (Single Instruction, Multiple Data) hardware capabilities for performance.

### Why Java is Not Purely Object-Oriented:

1. **Primitive Types**: Java includes primitive types like int, float, char, and boolean, which are not objects. They can exist independently of objects, which is not a typical characteristic of pure object-oriented languages.
2. **Static Methods**: Java supports static methods, which are methods associated with the class rather than an instance (object) of the class. This allows procedural-like programming within an object-oriented language.
3. **Encapsulation Limitations**: Java enforces encapsulation through syntax, but it is not absolute. Variables and methods can be accessed in ways that aren't fully restricted to object-oriented principles.

### Java Program Execution:

1. **Source Code and File Naming**: Java source code is written in a file with a .java extension. For example, a class PremProg should be saved as PremProg.java.
2. **Compilation**: The Java file is compiled using the javac command, generating a .class file containing bytecode.

bash

Copy code

javac PremProg.java

1. **Execution**: After compilation, the bytecode can be run on the Java Virtual Machine (JVM) using the java command, omitting the file extension:

bash

Copy code

java PremProg

### Types of Instructions in Java:

1. **Declaration Instructions**: Provide information to the compiler for code translation, such as variable declarations.
2. **Executable Instructions**: Can be divided into:
   * **Simple Instructions**: End with a semicolon (;), e.g., variable assignments.
   * **Structured Instructions**: Include control flow structures like if, for, while.
   * **Blocks**: Grouped instructions enclosed in curly braces ({}).

### Floating-Point Operations in Java:

* **No Execution Halts**: Java follows IEEE 754 standards for floating-point arithmetic, meaning operations (including division by zero) do not cause execution to halt. Special values like positive infinity, negative infinity, and "Not-a-Number" (NaN) are used to represent undefined or infinite results.

### Short-Circuit Operators (&& and ||):

* **Behavior**: The && and || operators only evaluate the second operand if necessary:
  + Example:

java

Copy code

a < b && c < d

Here, c < d is only evaluated if a < b is true.

* **Non-Short-Circuit Operators**: The single & and | operators always evaluate both operands, as does ^ (exclusive OR).
* **Practical Example**:

java

Copy code

if (i < max && t[i++] != 0)

In this code, i++ (incrementation of i) happens only if i < max is true. In contrast, with &, the increment happens regardless of the result of i < max:

java

Copy code

if (i < max & t[i++] != 0)

### Implicit Legal Conversions in Java:

* **Numeric Type Hierarchy**:
  + byte → short → int → long → float → double
  + char → int → long → float → double

### Avoiding Implicit Conversions:

Consider the following code:

java

Copy code

byte b;

b = b + 1; // Compilation error: b+1 is an int, and cannot be assigned to a byte

b++; // OK

* The ++ operator does not apply any implicit conversion to its operand, so b++ works correctly, but b = b + 1 results in an error because the expression b + 1 is evaluated as an int.
* To fix the issue, an explicit cast must be used:

java

Copy code

b = (byte) (b + 1);

### Conversion with += Operator:

* The += operator allows implicit conversion even when the hierarchy of types is not respected. It behaves as if an explicit cast has been applied:

java

Copy code

b += 1; // Works even though 1 is an int, due to implicit conversion.

### Forced Conversion Using Casts:

* When a value can be represented in the destination type, a cast results in only a loss of precision (e.g., truncation in case of narrowing conversions).
* If the value is out of the destination type's range, the result may be nonsensical. For example, casting from floating-point to integer can lead to truncation, and conversion of large numbers results in the loss of significant bits.

### do-while Loop with No Operation:

java

Copy code

do ; while(condition);

This is a valid loop structure in Java, where the loop executes without performing any operation.

### for Loop:

* Java allows multiple expressions in the first (initialization) and third (update) sections of a for loop, separated by commas.
* The second part (condition) must always be a boolean expression.

Example:

java

Copy code

for (int i = 0, j = 0; i < 10; i++, j++) {

// Loop body

}

* The first part of the for loop can also be a variable declaration, but you cannot declare variables of different types in a single initialization:

java

Copy code

for (int i = 0, j = 0; i < 10; i++) {

// OK

}

java

Copy code

for (int i = 0, long j = 0; i < 10; i++) {

// Error: cannot declare variables of different types together

}

* In nested loops, the break statement exits the innermost loop.

### break with Labels (Similar to goto):

* Java provides labeled break statements to jump out of outer loops or blocks.
* The label must correspond to a loop or block containing the current statement.

Example:

java

Copy code

outer:

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

if (j == 5) break outer; // Break out of the outer loop

}

}

### continue with Labels:

* Unlike break, continue with a label must refer to a looping structure (a loop such as for, while, or do-while).

Example:

java

Copy code

outer:

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

if (j == 5) continue outer; // Skip to the next iteration of the outer loop

}

}

### Java Classes and Methods:

1. **Multiple Classes in One File**:
   * A Java source file can contain multiple classes, but only one class can be declared as public.
   * The class containing the main method **must** be public so the Java Virtual Machine (JVM) can access it.
   * A class with no access modifiers (default access) is accessible by all classes in the same package.
2. **Garbage Collection**:
   * There is no operator in Java to explicitly destroy an object. Java uses automatic garbage collection to manage memory by automatically freeing memory for objects that are no longer referenced.
3. **Local Variables in Blocks**:
   * Variables can be declared locally within a block (e.g., inside a method or loop). Their scope is limited to that block, and their memory is allocated upon entering the block and deallocated when exiting.
   * A local variable cannot share the same name as another variable from an outer block.
4. **Static Fields (Class Fields)**:
   * **Static fields** exist independently of any instances of the class. They are shared across all instances.
   * Declaration: Use the static keyword to declare a static field.

Example:

java

Copy code

static int count = 0; // Shared across all objects of the class

1. **Static Methods (Class Methods)**:
   * Static methods are invoked without needing to create an instance of the class. They can only operate on static fields and cannot access instance fields.
   * Static methods are useful for operations not tied to object state.

Example:

java

Copy code

static void displayCount() {

System.out.println(count); // Works because 'count' is static

}

1. **Initialization of Static Final Fields**:
   * A final static field must be initialized at the point of declaration because it cannot be assigned a value later in a constructor. Once assigned, its value cannot change.

Example:

java

Copy code

static final int MAX = 100; // Must be initialized during declaration

1. **Method Overloading**:
   * Java allows **overloading** methods, which means multiple methods can have the same name but different parameter types or counts.
   * **Method Selection Process**:
     1. If no method matches, there is a **compilation error**.
     2. If only one method matches, it is selected.
     3. If multiple methods match, the compiler chooses the "most specific" method by eliminating less appropriate ones. If ambiguity remains, a **compilation error** occurs.
2. **Data Passing in Methods**:
   * In Java, **objects are passed by reference**, but the reference itself is passed **by value**. This means the method gets a copy of the reference, and it can modify the object, but it cannot change the reference itself.
   * Modifications made to the object inside the method will affect the original object.
3. **this Keyword**:
   * In Java, this(...) can be used within a constructor to call another constructor in the same class, promoting code reuse and consistency in object initialization.

Example:

java

Copy code

class Point {

int x, y;

Point() {

this(0, 0); // Calls another constructor

}

Point(int x, int y) {

this.x = x;

this.y = y;

}

}

1. **Recursion**:
   * Recursive methods in Java are less efficient in terms of both time and memory than iterative methods. Recursion should be used when an iterative solution is not obvious.
   * Recursion involves a method calling itself until it reaches a base case.
2. **Object Composition (Member Object)**:
   * A member object in Java is an instance of another class that is part of a class (also called "has-a" relationship or **composition**).
   * Composition is different from inheritance, which follows the "is-a" relationship.

### Nested Classes

1. **Association with Outer Instance**:
   * Each object of an **inner class** is tied to a specific instance of the outer class that created it. This allows direct access to the non-static members of the outer class.
2. **Enhanced Encapsulation**:
   * Inner classes can access private members of the outer class, strengthening encapsulation and facilitating data handling.
3. **Practical Use**:
   * Inner classes are useful when a class is closely related to another, where careful control over visibility and encapsulation is necessary.
4. **Static Method Limitations**:
   * A static method in an outer class cannot create an instance of a non-static inner class, as static methods are not tied to any object.
   * Inner classes cannot contain static members.
5. **Creating Inner Class Instances from Outside**:
   * When outside the outer class, you can declare a reference to an object of the inner class, but you must instantiate it by associating it with an instance of the outer class.

java

Copy code

E e = new E();

E.I i = e.new I(); // Creating an inner class object linked to the outer class object

1. **Local Inner Classes**:
   * You can define an inner class inside a method of an outer class. In this case, objects of the inner class can only be instantiated within that method. The inner class can access final local variables of the method.
2. **Static Inner Classes**:
   * When instantiating a static inner class from outside the outer class, no outer class instance is needed. Static inner classes do not have access to the non-static members of the outer class but can access its static members.

java

Copy code

E.I i = new E.I(); // No need for an instance of the outer class

1. **Visibility of Inner Classes**:
   * If an inner class is declared public, it is accessible wherever the outer class is accessible.
   * If declared private, it is only accessible within the outer class.
   * With no visibility modifier, it is accessible only within the same package.

### Packages

1. **Organizing Classes**:
   * **Packages** allow for logical grouping of classes under a common identifier, making it easier to manage large software projects.
2. **Default Access**:
   * If no visibility modifier (like public, protected, or private) is specified, the class is only accessible by other classes in the same package.

### Arrays

1. **Jagged Arrays**:
   * Java allows **jagged arrays**, where rows can have different lengths. For example:

java

Copy code

int[][] arr = new int[3][];

arr[0] = new int[2]; // Row 1 with 2 columns

arr[1] = new int[3]; // Row 2 with 3 columns

### Inheritance

1. **Default Constructors**:
   * If neither the base class nor the derived class has explicit constructors, Java provides a default constructor for both. If a no-argument constructor is needed in the base class but not provided, this results in a compilation error.
2. **Method Overriding (@Override)**:
   * Overriding occurs when a method in a derived class has the same signature as a method in the base class. The overridden method in the derived class replaces the base class method for instances of the derived class.
   * The @Override annotation is used to explicitly indicate method overriding.
3. **Using super**:
   * The keyword super is used to call methods or constructors of the base class from within the derived class.
4. **Overloading Methods**:
   * **Overloading** occurs when multiple methods have the same name but different signatures (i.e., differing in the number or types of parameters).
   * The appropriate method is selected based on the method signature during execution.
5. **Method Overriding Rules**:
   * In **overriding** (where the method signatures are identical), the return type and access levels must be the same or more permissive than those in the base class.
   * The access level of a method cannot be more restrictive in the derived class than in the base class.
   * A method in the derived class can have more permissive access (e.g., changing private to public).
6. **Method Overriding Restrictions**:
   * If a method in a derived class has the same name as one in a base class but a different signature, it's considered **overloading**, not overriding, and both methods can coexist.
   * Methods marked as static cannot be overridden in derived classes.
7. **Fields in Inheritance**:
   * When a field is redeclared in a derived class, the original field in the base class is not overridden. Both fields exist, but only the derived class field is visible from outside.

### Access Control Levels in Java

1. **Access Control Levels**:
   * **private**: Accessible only within the class.
   * **Package access** (no modifier): Accessible only within the same package.
   * **protected**: Accessible within the same package and by subclasses.
   * **public**: Accessible from anywhere.
2. **Overriding Access Rules**:
   * An overridden method in a derived class cannot have more restrictive access than the method in the base class, but it can have more permissive access.

### Polymorphism

1. **Implicit Conversion and Assignment Compatibility**:
   * Java allows assigning a reference of a derived class to a variable of its parent class. This enables polymorphism, where a parent type variable can hold a reference to a child object.
2. **Method Resolution with Polymorphism**:
   * **Compilation**: The best method signature matching the call is selected based on the declared type of the variable.
   * **Execution**: The actual method called is determined by the runtime type of the object, allowing dynamic method dispatch (polymorphic behavior).
3. **instanceof Keyword**:
   * This keyword checks if an object is an instance of a specific class or interface at runtime.
4. **super and Dynamic Binding**:
   * The super keyword bypasses polymorphism and is not subject to dynamic binding. It directly refers to methods in the parent class.
5. **Polymorphism in Arrays**:
   * An array of a derived type can be assigned to an array of a parent type. For example, if B is a subclass of A, an array of B can be treated as an array of A. However, this only applies to objects, not primitive types.

### Class Object

* All Java classes implicitly inherit from the Object class. For a method to be called on an object, the method must either be defined in the object's class or inherited from a superclass (including Object).

### Protected Members

* Members declared as protected are accessible to classes in the same package and to subclasses, whether they are in the same package or not. This flexibility can complicate design, making protected less commonly used in practice.

### Final Classes and Methods

1. **Final Classes**:
   * A class declared as final cannot be extended (inherited).
   * A **final class** is not the same as a class where all methods are final.
2. **Final Methods**:
   * A final method cannot be overridden in a subclass.
3. **Final vs. Abstract**:
   * A final class is concrete and cannot be extended, while an abstract class is meant to be extended and cannot be instantiated directly.

### Abstract Classes

1. **Abstract Methods**:
   * If a class contains any abstract method, it must be declared as abstract, even if not explicitly indicated by the abstract keyword.
   * An abstract method must be public because it is intended to be overridden in subclasses.
   * Argument names in abstract method headers are required but are not functional at this stage.
2. **Abstract Classes and Derived Classes**:
   * A derived class of an abstract class does not need to implement all abstract methods; if it doesn’t, it remains abstract.
   * A non-abstract base class can have an abstract subclass.

### Interfaces

1. **Interface Definition**:
   * An interface provides method declarations that must be implemented by any class that implements the interface.
   * Interfaces can have the same access modifiers as classes (public, protected, etc.).
2. **Class Implementation**:
   * A class that implements an interface must override and provide implementations for all of its methods.
3. **Interface Characteristics**:
   * **Instantiation**: Interfaces cannot be instantiated directly, but you can instantiate objects that implement an interface or use anonymous classes and lambda expressions for this purpose.
   * **Polymorphism**: A variable with a reference to an interface can refer to objects of any class that implements that interface.
4. **Multiple Interfaces**:
   * A class can implement multiple interfaces.
   * If two interfaces have methods with the same name but different return types, a class cannot implement both interfaces.
5. **Interface Constants**:
   * Methods in interfaces need to be implemented, while constants (fields) are static and final, meaning they are constants accessible without needing an instance of the class.
6. **Interface Inheritance**:
   * Interfaces can inherit from other interfaces, concatenating method declarations.
7. **Cloneable Interface**:
   * A class that implements the Cloneable interface can undergo deep cloning using the clone() method, either provided by Object or by an overriding method within the class itself.

### Classes Enveloppes (Wrapper Classes)

* Java provides **wrapper classes** for each of the primitive types: Boolean, Byte, Character, Short, Integer, Long, Float, and Double. These classes encapsulate primitive values in an object, allowing primitives to be treated like objects.
* Wrapper classes allow primitive values to be used where objects are required (e.g., in collections).
* Each wrapper class provides methods like xxxValue() to extract the primitive value (xxx represents the primitive type, such as intValue() for Integer).

### Anonymous Classes

* **Anonymous classes** are a way to define a class without a name and instantiate it in a single expression.
* They are **non-reusable**, as they don’t have a name and are often used in situations where a class is only needed once, like providing a quick implementation for an interface or subclass.
* **Limitations**:
  + An anonymous class cannot define any new methods beyond those inherited or overridden.
  + It is commonly used for event handling, callback functions, or short-lived objects.

### Strings in Java

* Strings in Java are **immutable**, meaning they cannot be changed once created.
* Concatenation with the += operator only works if at least one operand is a string.
* String comparison with == checks for **reference equality** (i.e., if two string objects are the same in memory), not value equality. To compare values, use .equals().

#### Common String Methods:

* charAt(int index): returns the character at the specified index.
* length(): returns the length of the string.
* indexOf(), lastIndexOf(): find the position of characters or substrings.
* equals(), equalsIgnoreCase(): compare strings for value equality.
* compareTo(): compares two strings lexicographically.
* replace(): replaces occurrences of a character or substring with another.
* substring(): extracts a portion of the string.
* toLowerCase(), toUpperCase(): convert the string to lower or upper case.
* trim(): removes leading and trailing whitespace.
* String.valueOf(): converts different data types to strings.

#### Parsing Methods for Wrappers:

* Integer.parseInt(), Double.parseDouble(), etc.: convert strings to their respective primitive types.

#### StringBuffer

* StringBuffer is a mutable alternative to String that allows strings to be modified.
* **Key Methods**:
  + append(): adds content to the end.
  + insert(): inserts content at a specific index.
  + replace(): replaces a portion of the string.
  + setCharAt(): modifies a character at a specific index.
  + charAt(): accesses a character.
  + toString(): converts the StringBuffer object back to a String.
  + ensureCapacity(): optimizes memory allocation for large buffers, reducing reallocations.

### Exceptions

* **Custom Exceptions**: You can define custom exception classes by extending Exception:

java

Copy code

class MonExcep extends Exception {

public MonExcep(String message) {

super(message);

}

}

* + Custom exceptions can be thrown with throw new MonExcep("Error message").
* **Exception Handling**:
  + try-catch-finally blocks allow handling exceptions.
  + **catch**: Used to handle the exception. You can access the exception object inside the catch block using e.getMessage().
  + **finally**: This block always executes, whether an exception occurs or not. It is often used for cleanup activities like closing resources.
* **Exception Propagation**:
  + If a method does not handle an exception, it must declare it using throws in its method signature. The exception propagates back to the calling method until it is caught.
* **Error Handling Considerations**:
  + Be mindful of the inheritance hierarchy in exception handling. Catch blocks should handle exceptions from more specific to more general.

#### Throwable Class

* Exception and Error both derive from Throwable, but they serve different purposes:
  + Exception: Typically represents conditions that a program should handle.
  + Error: Represents serious problems that an application usually should not catch (e.g., OutOfMemoryError). These are often unrecoverable.

### Threads in Java

#### Creating Threads

1. **Using Thread Class**:  
   To create a thread by extending the Thread class:

java

Copy code

class MyThread extends Thread {

public void run() {

// Code to be executed by the thread

}

}

MyThread t = new MyThread();

t.start(); // Starts the thread

* + The start() method triggers the execution of the run() method in a new thread.

1. **Using Runnable Interface**:  
   To create a thread by implementing the Runnable interface:

java

Copy code

class MyRunnable implements Runnable {

public void run() {

// Code to be executed by the thread

}

}

MyRunnable e1 = new MyRunnable();

Thread t1 = new Thread(e1); // Create a thread associated with the Runnable object

t1.start(); // Starts the thread

* + This approach allows flexibility because the class can extend another class, as Java does not support multiple inheritance.

Additionally, you can define a custom start() method in the class implementing Runnable:

java

Copy code

public void start() {

Thread t = new Thread(this);

t.start();

}

#### Interrupting Threads

* Java allows one thread to signal another thread to stop by using the interrupt() method:

java

Copy code

t1.interrupt(); // Request to interrupt the thread

* + The interrupt() method doesn't directly stop the thread but sets an interrupt flag.
  + **Checking Interrupt Status**:
    - Use Thread.interrupted() to check if the current thread has been interrupted.
    - Use isInterrupted() to check if another thread's interrupt flag is set without clearing it.
  + **Thread Groups**: Multiple threads can be grouped together using ThreadGroup, and all threads in a group can be interrupted using interrupt() on the group.

#### Types of Threads

* **User Threads**: Standard threads.
* **Daemon Threads**: Background threads that run as services. They can be set using setDaemon(true) before starting the thread.

#### Thread Coordination

1. **Thread Synchronization** (Handling Concurrency):
   * **Problem 1**: Two threads should not access the same object at the same time.
   * **Problem 2**: A thread must wait for another to finish a task before it can proceed.

**Synchronized Methods**:

* + Declaring methods as synchronized ensures that only one thread can execute that method on an object at a time.

java

Copy code

public synchronized void someMethod() {

// Code that needs to be synchronized

}

* + A thread acquires a lock on the object when calling a synchronized method, and other synchronized methods cannot be called on that object until the lock is released.

**Synchronized Blocks**:

* + If only a part of the method needs synchronization, use a synchronized block:

java

Copy code

synchronized (someObject) {

// Critical section

}

1. **Wait and Notify**:
   * **Problem 2**: A thread must wait for another to finish working on an object before it can proceed.

**Wait and Notify**:

* + wait(): Makes a thread wait, releasing the lock on the object until notified.
  + notify(): Wakes up a single waiting thread.
  + notifyAll(): Wakes up all waiting threads.

java

Copy code

synchronized (someObject) {

someObject.wait(); // Wait until notified

}

synchronized (someObject) {

someObject.notifyAll(); // Notify all waiting threads

}

#### Thread States

1. **New**: Thread is created but not yet started.
2. **Runnable**: Thread is ready to run but not necessarily running.
3. **Running**: Thread is actively running.
4. **Blocked/Waiting**: Thread is waiting for a resource or notification.
5. **Sleeping**: A thread can be put to sleep using sleep().
6. **Terminated**: Thread has completed its execution or has been stopped.

* **Thread Transition Actions**:
  + **Yield**: The yield() method allows a thread to relinquish the CPU, but it may be chosen again.
  + **Sleep**: A thread can be made to sleep for a specified period using sleep().
  + **Wait**: A thread can wait for a signal (wait()).
  + **Exit**: The thread finishes (System.exit()).

#### Thread Priorities

* **Thread Priorities** range from MIN\_PRIORITY (1) to MAX\_PRIORITY (10), with NORM\_PRIORITY (5) as the default.
* **Priority Handling**:
  + The thread with the highest priority is generally selected for execution when ready.
  + The priority of a thread can be changed using setPriority(int priority), but modifying priorities is not recommended for portability.

### Conclusion

* Java provides robust tools for managing multi-threaded applications, offering multiple ways to create threads, interrupt them, and synchronize their actions. The use of synchronized methods or blocks, combined with wait() and notifyAll(), helps manage concurrency and ensures threads coordinate properly.