Ans 1, (a): Java and C++ are both powerful, high-level programming languages, but they have several differences in terms of their syntax, design, and execution. Here's a brief comparison:

1. Language Paradigm:

- Java is an object-oriented, class-based language with support for multi-threading. It is designed to be platform-independent.

- C++ is a multi-paradigm language that supports both procedural and object-oriented programming. It is generally platform-dependent.

2. Memory Management:

- Java uses automatic memory management through garbage collection, which makes it less error-prone in terms of memory leaks.

- C++ allows manual memory management, which gives developers more control but can lead to memory-related bugs if not handled properly.

3. Compile vs. Interpretation:

- Java is typically compiled into bytecode, which is then executed by the Java Virtual Machine (JVM) at runtime. This allows it to be platform-independent.

- C++ is compiled directly into machine code for a specific platform.

4. Pointers:

- Java does not have explicit pointer support, which can make it safer but may limit certain low-level operations.

- C++ provides full support for pointers, allowing more direct memory manipulation but also making it easier to introduce bugs.

5. Standard Libraries:

- Java has a robust standard library with built-in support for data structures and various APIs for network communication, GUI development, etc.

- C++ also has a standard library, but it's more lightweight compared to Java, and developers often rely on third-party libraries.

The JRE is a crucial component in the execution of Java applications. Here's how it works:

1. Compilation: Java source code is first compiled into bytecode by the Java compiler (javac). This bytecode is platform-independent.

2. JRE Installation: Users need to have the JRE installed on their system. The JRE includes the JVM and essential libraries required for Java applications to run.

3. Execution: When you run a Java application, the JRE comes into play. It loads the bytecode and passes it to the JVM.

4. Just-In-Time Compilation (JIT): The JVM interprets the bytecode and, in some cases, performs Just-In-Time compilation, converting bytecode to native machine code for the specific platform. This makes Java applications more efficient as they run.

5. Execution Environment: The JRE provides an execution environment, including memory management, thread handling, and access to standard libraries. It ensures that the Java application can run consistently on different platforms.

Ans 1, (b): In Java, methods play a crucial role in defining the behavior and functionality of objects. They are blocks of code that perform specific tasks or operations. Here's an overview of their role:

1. Encapsulation: Methods allow you to encapsulate the behavior of objects, meaning you can define what an object can do and how it does it without exposing its internal details.

2. Code Reusability: Methods enable code reuse by allowing you to define a set of instructions that can be called multiple times from different parts of your program.

3. Modularity: Methods promote modularity by breaking down a complex program into smaller, more manageable pieces, making it easier to develop and maintain.

4. Abstraction: Methods can provide an abstract interface to an object, allowing you to interact with it without needing to know its internal workings.

'this'keyword is a reference to the current instance of the class. It is typically used to disambiguate between instance variables and method parameters or local variables when they have the same name. For example:

public class MyClass {

private int myVariable;

public void setMyVariable(int myVariable) {

this.myVariable = myVariable; // 'this' is used to refer to the instance variable

}

}

In this example, 'this' is used to distinguish the instance variable `myVariable` from the method parameter `myVariable`. This helps prevent naming conflicts and ensures that the instance variable is correctly assigned.

Ans 2, (a): The `String` class in Java is used to represent and manipulate strings of characters. It is one of the most commonly used classes in Java and provides various methods to work with strings. Here's an explanation of three of its methods:

1. `length()`: The `length()` method is used to determine the number of characters in a `String`. It returns an integer representing the length of the string. For example:

String text = "Hello, World!";

int length = text.length(); // length will be 13

2. `charAt(int index)`: The `charAt(int index)` method is used to retrieve the character at a specific index within the string. Indices are zero-based, so the first character is at index 0. For example:

String text = "Java";

char firstChar = text.charAt(0); // firstChar will be 'J'

char thirdChar = text.charAt(2); // thirdChar will be 'v'

3. `substring(int beginIndex)`: The `substring(int beginIndex)` method is used to extract a portion of the string starting from the specified `beginIndex` to the end of the string. It returns a new string containing the characters from the specified index to the end. For example:

String text = "Programming";

String subString = text.substring(3); // subString will be "gramming"

These are just a few of the many methods provided by the `String` class in Java for string manipulation. Strings in Java are immutable, meaning that once a `String` object is created, its value cannot be changed. Instead, operations on strings typically return new `String` objects with the desired modifications.

Ans 2, (b): Exceptions in Java are a mechanism used to handle unexpected or exceptional situations that can occur during the execution of a program. These exceptional situations can include things like errors, runtime issues, or unexpected conditions that disrupt the normal flow of a program.

The main reasons for using exceptions in Java are:

1. Error Handling: Exceptions provide a way to handle errors and unexpected issues in a structured manner. This helps prevent a program from crashing or behaving unpredictably when it encounters a problem.

2. Separation of Concerns: Exceptions allow you to separate error-handling code from the main logic of your program, making the code more organized and easier to maintain.

3. Robustness: By handling exceptions, you can write code that can gracefully recover from errors or at least report them, enhancing the robustness of your software.

In Java, you handle exceptions using try-catch blocks. Here's how it works:

try {

// Code that might throw an exception

} catch (ExceptionType e) {

// Code to handle the exception

}

- The code that might throw an exception is placed within the `try` block.

- If an exception occurs, it's caught by the corresponding `catch` block. The `ExceptionType` should match the type of exception that you expect.

- You can have multiple catch blocks to handle different types of exceptions.

- You can also include a `finally` block to contain code that should always run, whether an exception is thrown or not.

For example:

try {

int result = 10 / 0; // This will throw an ArithmeticException

} catch (ArithmeticException e) {

System.out.println("An error occurred: " + e.getMessage());

} finally {

System.out.println("This will always execute.");

}

By using try-catch blocks, you can gracefully handle exceptions, report errors, and ensure your program's stability even when unexpected issues arise.

Ans 3, (a): Constructor overloading and method overriding are two distinct concepts in Java, each serving a different purpose.

Constructor Overloading:

- Constructor overloading is a feature in Java that allows you to define multiple constructors for a class with the same name but different parameter lists.

- The constructors have different sets of parameters, which provides flexibility when creating objects of the class.

- The choice of constructor to be used is based on the number and types of arguments passed when creating an object of the class.

Example of constructor overloading:

public class MyClass {

private int value;

// Constructor with no arguments

public MyClass() {

this.value = 0;

}

// Constructor with one argument

public MyClass(int value) {

this.value = value;

}

// Constructor with two arguments

public MyClass(int value1, int value2) {

this.value = value1 + value2;

}

}

Method Overriding:

- Method overriding occurs in the context of inheritance, where a subclass provides a specific implementation for a method that is already defined in its superclass.

- The overriding method in the subclass has the same name, return type, and parameters as the method in the superclass.

- Method overriding is used to achieve polymorphism and dynamic method dispatch.

Example of method overriding:

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks");

}

}

Key differences between constructor overloading and method overriding:

1. Constructor overloading involves multiple constructors in the same class with different parameter lists, while method overriding involves a subclass providing a specific implementation for a method defined in its superclass.

2. In constructor overloading, the constructors are called based on the arguments used when creating objects of the class. In method overriding, the method to execute is determined at runtime based on the actual object's type.

3. Constructors do not have a return type, while overridden methods must have the same return type in method overriding.

In summary, constructor overloading allows a class to have multiple constructors with different parameter lists for object creation, while method overriding is about providing a specific implementation of a method in a subclass to achieve polymorphism.

Ans 3, (b): In Java, the `final` keyword is used to indicate that something is constant and cannot be changed. Here are a few common uses of the `final` keyword:

1. Final Variables: When you declare a variable as `final`, its value cannot be modified once it's initialized. For example:

final int x = 10;

2. Final Methods: When you declare a method as `final` in a class, it means that the method cannot be overridden by subclasses.

3. Final Classes: A `final` class cannot be extended or subclassed. This is often used for classes where you want to prevent any further modification or specialization.

4. Final Parameters: When you declare method parameters as `final`, it means the values of these parameters cannot be changed within the method.

Regarding "Object array," it generally refers to an array in Java where the elements are of the `Object` class or are references to objects of any class. This allows you to create an array that can store objects of different types. Here's an example:

// Create an array of Object

Object[] objectArray = new Object[3];

// Initialize elements with objects of different types

objectArray[0] = "Hello, World"; // String

objectArray[1] = 42; // Integer

objectArray[2] = new Person("Alice"); // Custom class (Person)

// Access and retrieve objects from the array

String str = (String) objectArray[0];

int number = (int) objectArray[1];

Person person = (Person) objectArray[2];

In this example, `objectArray` is an array of type `Object`, and you can store objects of various types within it. When retrieving objects from the array, you often need to cast them to their original types because the array is of type `Object`. However, be cautious with such arrays, as they can lead to type casting issues and lack compile-time type safety. It's generally recommended to use specific data types or classes whenever possible to maintain type safety in your code.

Ans 4, (a): In Java, a 2D array is an array of arrays, essentially creating a grid-like structure. Each element in a 2D array can be accessed using two indices - one for the row and one for the column.

Here's a Java program to print the sum of all positive integer values in a 2D array:

public class SumOfPositiveValuesIn2DArray {

public static void main(String[] args) {

int[][] array = {

{1, -2, 3},

{4, -5, 6},

{-7, 8, -9}

};

int sum = 0;

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

for (int j = 0; j < array[i].length; j++) {

if (array[i][j] > 0) {

sum += array[i][j];

}

}

}

System.out.println("Sum of positive values in the 2D array: " + sum);

}

}

In this example, we have a 2D array called `array`, and we iterate through each element in the array. If the element is positive (greater than 0), we add it to the `sum` variable. Finally, we print the sum of positive values.

Ans 4, (b): Instance variables and class variables (also known as static variables) are two different types of variables in Java.

Instance Variable:

1. An instance variable is associated with an instance of a class, meaning each object of the class has its own copy of the instance variable.

2. Changes to the instance variable in one object do not affect the instance variable in another object.

3. Instance variables are declared without the `static` keyword and are typically used to store unique properties of each object.

Here's an example:

class Car {

String color; // instance variable

Car(String color) {

this.color = color;

}

}

public class InstanceVariableExample {

public static void main(String[] args) {

Car car1 = new Car("Red");

Car car2 = new Car("Blue");

System.out.println("Car 1 color: " + car1.color); // Red

System.out.println("Car 2 color: " + car2.color); // Blue

}

}

In this example, the `color` variable is an instance variable, and each `Car` object has its own `color`.

Class Variable (Static Variable):

1. A class variable is associated with the class itself, not with specific instances of the class. All objects of the class share the same class variable.

2. Changes to the class variable in one object will affect the class variable for all objects of the class.

3. Class variables are declared with the `static` keyword and are typically used for properties shared among all objects of the class.

Here's an example:

class Student {

static int totalStudents = 0; // class variable

Student() {

totalStudents++;

}

}

public class ClassVariableExample {

public static void main(String[] args) {

Student student1 = new Student();

Student student2 = new Student();

System.out.println("Total students: " + Student.totalStudents); // 2

}

}

In this example, `totalStudents` is a class variable, and it's shared among all `Student` objects.

Regarding static and non-static blocks, here's a Java program to demonstrate their working:

class BlockExample {

static {

System.out.println("This is a static block.");

}

{

System.out.println("This is a non-static block.");

}

BlockExample() {

System.out.println("This is a constructor.");

}

}

public class StaticAndNonStaticBlocks {

public static void main(String[] args) {

System.out.println("Creating object 1:");

BlockExample obj1 = new BlockExample();

System.out.println("Creating object 2:");

BlockExample obj2 = new BlockExample();

}

}

In this program, you'll see that the static block is executed when the class is loaded, while the non-static block is executed when an object of the class is created, just before the constructor.

Ans 5, (a): In Java, command line arguments are the values or parameters passed to a Java program when it's executed from the command line or terminal. These arguments are stored as strings in the `args` parameter of the `main` method.

Here's a Java program that demonstrates the use of a wrapper class to calculate and print the sum of positive integers passed as command line arguments:

public class SumPositiveIntegers {

public static void main(String[] args) {

int sum = 0;

for (String arg : args) {

try {

int num = Integer.parseInt(arg);

if (num > 0) {

sum += num;

}

} catch (NumberFormatException e) {

// Ignore non-integer command line arguments

}

}

System.out.println("Sum of positive integers: " + sum);

}

}

You can run this program from the command line by passing positive integers as arguments, like this:

java SumPositiveIntegers 5 10 15

This will calculate the sum of positive integers (5, 10, 15 in this example) and print the result. Non-integer arguments and negative numbers are ignored.

Ans 5, (b): The `char` data type in Java is used to represent a single 16-bit Unicode character. It can be used to store and manipulate individual characters, such as letters, numbers, symbols, and special characters. Some common uses of the `char` data type include processing text, creating user interfaces, and working with character-based data.

Here's an example of overriding the `toString()` method from the `Object` class in a custom Java class:

class MyObject {

private String data;

public MyObject(String data) {

this.data = data;

}

// Override the toString() method

@Override

public String toString() {

return "MyObject with data: " + data;

}

}

public class ObjectMethodOverride {

public static void main(String[] args) {

MyObject obj = new MyObject("Hello, World!");

// When you print obj, it will use the overridden toString() method

System.out.println(obj);

}

}s

In this example, the `MyObject` class overrides the `toString()` method inherited from the `Object` class. When you print an instance of `MyObject`, it will use the custom implementation of `toString()`, allowing you to define how the object should be represented as a string.

Keep in mind that `toString()` is just one of the methods you can override from the `Object` class. Depending on your requirements, you might override other methods such as `equals()`, `hashCode()`, or `clone()`.