

Question 1 Prompt

Does Java support passing a function as an argument to a function? What is the syntax?

Response

Yes, Java has grown to support functional programming features, especially after the introduction of Java 8. One of the main features is the introduction of lambda expressions and functional interfaces. Using these, we can indeed pass functions as arguments to other functions.

Here's how you can do it:

1. **Functional Interfaces:** Java introduced a new package `java.util.function` which contains many interfaces you can use to pass functions. Examples include `Predicate<T>`, `Function<T, R>`, `Consumer<T>`, `Supplier<T>`, and many more.
2. **Lambda Expressions:** This is a new syntax to define an instance of a functional interface without defining a separate named class.

Here's a quick example to illustrate:

Let's say you want to apply a function to an integer list:

```
```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

public class FunctionExample {
 public static void main(String[] args) {
 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

 // Function that doubles a number
 Function<Integer, Integer> doubleFunction = n -> n * 2;

 List<Integer> doubledNumbers = applyFunctionToList(numbers, doubleFunction);

 System.out.println(doubledNumbers); // Prints: [2, 4, 6, 8, 10]
 }

 public static List<Integer> applyFunctionToList(List<Integer> list, Function<Integer, Integer>
function) {
 for (int i = 0; i < list.size(); i++) {
 list.set(i, function.apply(list.get(i)));
 }
 return list;
 }
}
```
```

In this example:

- We have a list of integers.
- We defined a lambda expression `doubleFunction` that implements the `Function` interface to double an integer.
- We passed this `doubleFunction` to the `applyFunctionToList` method, which applies the function to each element of the list.

There are numerous possibilities and combinations with the new functional interfaces and lambda expressions which make Java much more expressive in terms of functional programming.

Question 2 Prompt

Consider the code

```
System.out.print(s==s1+" "+s.equals(s1));
```

I expected it to display two boolean values, but it displays only one.

Explain why?

Response

Let's dissect the expression:

```
```java
System.out.print(s == s1 + " " + s.equals(s1));
```
```

The `+` operator has higher precedence than the `==` operator. So the expression is evaluated as:

```
```java
System.out.print(s == (s1 + " " + s.equals(s1)));
```
```

Here's the step-by-step breakdown:

1. The method `s.equals(s1)` returns a `boolean` value (either `true` or `false`).
2. The `boolean` value from `s.equals(s1)` is concatenated with a space (`" "`) to produce a `String`. In Java, when you concatenate a `String` with a `boolean`, the `boolean` value gets automatically converted to its string representation (`"true"` or `"false"`).
3. The resultant `String` from the previous step is concatenated with `s1` to produce another `String`.
4. Now, you're comparing the original string `s` with this concatenated string using the `==` operator. This comparison checks if both strings are the same object in memory (not their content).

So the entire expression evaluates to a single `boolean` value, which is why you see only one result (either `true` or `false`).

If you want to print both results (from the `==` comparison and the `.equals()` method), you should break down the statement:

```
```java
System.out.print(s == s1);
System.out.print(" ");
System.out.println(s.equals(s1));
```
```

This will print two boolean values, separated by a space.

Question 3 Prompt

Non-abstract methods have an implementation. The same applies to the default methods.

Non-abstract methods are inherited and can be overwritten. The same applies to default methods.

What is the difference between default methods and non-abstract ones?

Answer briefly.

Response

The primary difference between default methods and non-abstract methods lies in the context in which they are used and their purpose:

1. **Definition Context**:

- **Default Methods**: Defined in interfaces. Introduced in Java 8 to allow adding new methods to interfaces without breaking existing implementations of those interfaces.
- **Non-abstract Methods**: Defined in classes.

2. **Purpose**:

- **Default Methods**: Provide a default implementation for a method in an interface. This means that classes implementing the interface can use the default method without providing an implementation. However, they can also override the default method.
- **Non-abstract Methods**: Regular class methods that can be inherited by subclasses. Subclasses can override these methods.

In summary, while both default and non-abstract methods provide implementations and can be overridden, default methods are specific to interfaces and serve to provide backward compatibility, whereas non-abstract methods are standard methods in classes.

PF Prompt #1

Develop a Java function that calculates the power of a real number raised to an integer exponent.

Specifications

1. Interface: `public static double pow(double b, int e)`
2. Don't use `Math.pow` or `BigDecimal.pow`
3. Achieve the maximum possible precision

Response

Certainly! To raise a real number `b` to an integer exponent `e`, the algorithm can be optimized using the "exponentiation by squaring" method. This approach reduces the number of multiplications by breaking down the problem into smaller parts.

Here's a possible solution:

```
```java
public class PowerCalculator {

 public static void main(String[] args) {
 System.out.println(pow(2, 10)); // 1024.0
 System.out.println(pow(2, -3)); // 0.125
 System.out.println(pow(5, 0)); // 1.0
 System.out.println(pow(1.5, 3)); // 3.375
 }

 public static double pow(double b, int e) {
 // If the exponent is zero, return 1 (because anything raised to 0 is 1)
 if (e == 0) {
 return 1.0;
 }

 // If the exponent is negative, invert the base and make the exponent positive
 if (e < 0) {
 e = -e;
 b = 1 / b;
 }

 double result = 1.0;
```

```

while (e > 0) {
 // If the current exponent is odd, multiply the result by the base
 if ((e & 1) == 1) {
 result *= b;
 }

 // Square the base and halve the exponent for the next iteration
 b *= b;
 e >>= 1;
}

return result;
}
}
...

```

This algorithm takes advantage of the fact that the exponentiation of a number can be broken down into smaller multiplications. The while-loop iterates only through the number of bits in the integer representation of `e`, making it faster than a naive approach of simply multiplying the base number repeatedly.

### PF Prompt #2

Can you improve the precision of your function? I checked it against `Math.pow` and found significant discrepancies.

Examples:

base= 502 exponent= 9 GPT.pow= 2.0245730632526733E24 Math.pow= 2.024573063252673E24  
 difference= 2.68435456E8

base= 504 exponent= 9 GPT.pow= 2.098335016107156E24 Math.pow= 2.0983350161071556E24  
 difference= 2.68435456E8

### Response

I understand your concern. The discrepancies you've identified come from the limited precision of the `double` data type, especially for large values or repeated multiplications.

In the given approach, squaring the base and multiplying the result can introduce small errors at each step, which then accumulate. The built-in `Math.pow` method likely uses specialized algorithms or hardware features to maintain higher precision.

To improve precision, one strategy is to implement a decimal arithmetic library or to use `BigDecimal`. Here's a more precise version of the `pow` function using `BigDecimal`:

```

```java
import java.math.BigDecimal;
import java.math.MathContext;

public class PowerCalculator {

    public static void main(String[] args) {
        System.out.println(pow(2, 10));    // 1024.0
        System.out.println(pow(2, -3));    // 0.125
        System.out.println(pow(5, 0));     // 1.0
        System.out.println(pow(1.5, 3));   // 3.375

        double base = 502;
        int exponent = 9;
        double gptResult = pow(base, exponent);
        double mathResult = Math.pow(base, exponent);
    }
}

```

```

        System.out.println("gpt.pow= " + gptResult);
        System.out.println("Math.pow= " + mathResult);
        System.out.println("difference= " + (gptResult - mathResult));
    }

    public static double pow(double b, int e) {
        BigDecimal base = new BigDecimal(b, MathContext.DECIMAL128); // Using high precision context
        BigDecimal result = BigDecimal.ONE;
        BigDecimal baseSquared;

        if (e < 0) {
            base = BigDecimal.ONE.divide(base, MathContext.DECIMAL128);
            e = -e;
        }

        while (e > 0) {
            if ((e & 1) == 1) {
                result = result.multiply(base, MathContext.DECIMAL128);
            }
            baseSquared = base.multiply(base, MathContext.DECIMAL128);
            base = baseSquared;
            e >>= 1;
        }

        return result.doubleValue();
    }
}
...

```

Note that while this version should have better precision than the previous one, using `BigDecimal` might introduce a performance penalty.

TTT Prompt #1

Develop a command-line tic-tac-toe application consisting of the following classes:

Player, Board, LivePlayer, RBPlayer, and Game.

Player is an Abstract class containing
 final char id
 abstract method Board move(Board board)

Class Board represents the game board. It contains the following public function members:
 void displayBoard(): It displays the game board on its current status
 char win(): It returns the winner's id. If there is no winner, it returns a white character.

Class LivePlayer represents a human player. It is a concrete class implementation inherited from Player.

Class RBPlayer represents an artificial Rule-based Player. It is based on the following rules:
 A. If there is a movement to win, select it.
 B. If the opponent has a movement to win, select it to block the opponent from winning.

Game: Uses the above-described classes to implement a tic-tac-toe game.

Response

Below is a simple implementation of a command-line tic-tac-toe game with the described classes:

```
``java
// Abstract Player class
abstract class Player {
    final char id;

    Player(char id) {
        this.id = id;
    }

    abstract Board move(Board board);
}

class Board {
    private char[][] board;

    public Board() {
        board = new char[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                board[i][j] = ' ';
            }
        }
    }

    public char get(int x, int y) {
        return board[x][y];
    }

    public void set(int x, int y, char player) {
        board[x][y] = player;
    }

    public void displayBoard() {
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(board[i][j]);
                if (j < 2) {
                    System.out.print(" | ");
                }
            }
            System.out.println();
            if (i < 2) {
                System.out.println("-----");
            }
        }
    }

    public char win() {
        // Check rows, columns and diagonals
        for (int i = 0; i < 3; i++) {
            if (board[i][0] != ' ' && board[i][0] == board[i][1] && board[i][1] == board[i][2]) {
                return board[i][0];
            }
            if (board[0][i] != ' ' && board[0][i] == board[1][i] && board[1][i] == board[2][i]) {
                return board[0][i];
            }
        }
    }
}
```

```

    }
}
if (board[0][0] != ' ' && board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
    return board[0][0];
}
if (board[0][2] != ' ' && board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
    return board[0][2];
}
return ' ';
}
}

```

```

class LivePlayer extends Player {

```

```

    public LivePlayer(char id) {
        super(id);
    }

```

```

    @Override

```

```

    Board move(Board board) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        int x, y;
        do {
            System.out.println("Enter your move (row [0-2] and column [0-2]) separated by space:");
            x = scanner.nextInt();
            y = scanner.nextInt();
        } while (board.get(x, y) != ' ');
        board.set(x, y, this.id);
        return board;
    }
}

```

```

class RBPlayer extends Player {

```

```

    public RBPlayer(char id) {
        super(id);
    }

```

```

    @Override

```

```

    Board move(Board board) {
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board.get(i, j) == ' ') {
                    board.set(i, j, this.id);
                    if (board.win() == this.id) {
                        return board;
                    }
                    board.set(i, j, ' ');
                }
            }
        }
    }
    char opponentId = (this.id == 'X') ? 'O' : 'X';
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board.get(i, j) == ' ') {
                board.set(i, j, opponentId);
                if (board.win() == opponentId) {

```

```

        board.set(i, j, this.id);
        return board;
    }
    board.set(i, j, ' ');
}
}
}
// If no strategic move found, make the first available move
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (board.get(i, j) == ' ') {
            board.set(i, j, this.id);
            return board;
        }
    }
}
return board; // should never reach here in a valid game
}
}

```

```

public class Game {
    public static void main(String[] args) {
        Board board = new Board();
        Player player1 = new LivePlayer('X');
        Player player2 = new RBPlayer('O');

        Player current = player1;
        int moves = 0;
        while (true) {
            board.displayBoard();
            board = current.move(board);
            moves++;
            if (board.win() != ' ' || moves == 9) {
                break;
            }
            current = (current == player1) ? player2 : player1;
        }

        board.displayBoard();
        if (board.win() == 'X') {
            System.out.println("Player X wins!");
        } else if (board.win() == 'O') {
            System.out.println("Player O wins!");
        } else {
            System.out.println("It's a tie!");
        }
    }
}
...

```

Here's a breakdown of how the code works:

1. The `Board` class manages the state of the tic-tac-toe board.
2. The `Player` class is an abstract representation of a player. `LivePlayer` and `RBPlayer` are the two concrete implementations.
3. The `LivePlayer` class reads a move from the user.
4. The `RBPlayer` class makes decisions based on simple rule-based logic.

5. The `Game` class drives the gameplay using these classes.

When you run the `Game` class, it will initiate a tic-tac-toe match where the human player (represented as 'X') plays against the rule-based AI (represented as 'O'). The game continues until there's a winner or all spots on the board are filled.

TTT Prompt #2

Can you add the class MinimaxPlayer representing an artificial player based on the well-known minimax algorithm?

Response

Certainly! The Minimax algorithm is a recursive algorithm for choosing the next move in a game. In tic-tac-toe, the Minimax algorithm can be easily applied due to the game's simple nature.

Here's the `MinimaxPlayer` class:

```
```java
class MinimaxPlayer extends Player {

 public MinimaxPlayer(char id) {
 super(id);
 }

 @Override
 Board move(Board board) {
 int bestScore = Integer.MIN_VALUE;
 int bestMoveX = -1;
 int bestMoveY = -1;

 for (int i = 0; i < 3; i++) {
 for (int j = 0; j < 3; j++) {
 if (board.get(i, j) == ' ') {
 board.set(i, j, this.id);
 int score = minimax(board, 0, false);
 board.set(i, j, ' ');
 if (score > bestScore) {
 bestScore = score;
 bestMoveX = i;
 bestMoveY = j;
 }
 }
 }
 }

 board.set(bestMoveX, bestMoveY, this.id);
 return board;
 }

 private int minimax(Board board, int depth, boolean isMaximizing) {
 char winner = board.win();
 if (winner == this.id) return 10 - depth;
 if (winner != ' ' && winner != this.id) return depth - 10;
 if (isFull(board)) return 0;

 if (isMaximizing) {
 int bestScore = Integer.MIN_VALUE;
 for (int i = 0; i < 3; i++) {
```

```

 for (int j = 0; j < 3; j++) {
 if (board.get(i, j) == ' ') {
 board.set(i, j, this.id);
 int score = minimax(board, depth + 1, false);
 board.set(i, j, ' ');
 bestScore = Math.max(score, bestScore);
 }
 }
 }
 return bestScore;
} else {
 char opponentId = (this.id == 'X') ? 'O' : 'X';
 int bestScore = Integer.MAX_VALUE;
 for (int i = 0; i < 3; i++) {
 for (int j = 0; j < 3; j++) {
 if (board.get(i, j) == ' ') {
 board.set(i, j, opponentId);
 int score = minimax(board, depth + 1, true);
 board.set(i, j, ' ');
 bestScore = Math.min(score, bestScore);
 }
 }
 }
 return bestScore;
}
}

private boolean isFull(Board board) {
 for (int i = 0; i < 3; i++) {
 for (int j = 0; j < 3; j++) {
 if (board.get(i, j) == ' ') {
 return false;
 }
 }
 }
 return true;
}
}
...

```

You can add the `MinimaxPlayer` to your game by replacing the `RBPlayer` instantiation in the `Game` class with:

```

...java
Player player2 = new MinimaxPlayer('O');
...

```

The Minimax algorithm evaluates each possible move by assuming that the opponent will also play optimally. This method ensures that the AI is unbeatable in tic-tac-toe and will always either win or tie the game.

### TTT Prompt #2.1

No improvement. It's still straightforward for anyone to win your MinimaxPlayer. I'm giving you the game board if it can help you. Please don't give me the same wrong algorithm again. If you can't do better, just let me know.

Player X, enter your move (row [0-2] and column [0-2]):

2 0

-----  
| O | O | X |  
-----

| O | X | |  
-----

| X | | |  
-----

Player X wins!

## Response

### DA-E Prompt #1

The Code below fails with IndexOutOfBoundsException.

```
import java.util.ArrayList;
import java.util.List;
```

```
public class Debug2 {
 static ArrayList<String> l=new ArrayList<>();

 static void load() {
 l.add("Green");
 l.add("Black");
 l.add("Blue");
 l.add("White");
 l.add("Pink");
 l.add("Black");
 }

 static void delAll(List<String> l, String target) {
 int size=l.size();
 for (int i=0; i<size; i++)
 if (target.equals(l.get(i))) {
 l.remove(i);
 }
 }

 public static void main(String[] args) {
 load();
 delAll(l,"Black");
 }
}
```

Explain the error and correct the code.

## Response

The error in the code is due to how elements are removed from the `ArrayList` inside the `delAll` method.

When you remove an element from an `ArrayList` using its index, the elements that follow are shifted left. This changes the indexes of subsequent elements. Thus, if two "Black" elements are consecutive in the list, the code will only remove the first one and skip the second because the loop increment (`i++`) and the removal (`l.remove(i)`) conflict.

For example:

Let's assume `l` is ["Green", "Black", "Black", "Blue"].

When `i=1`, "Black" at index 1 is removed, making `l` ["Green", "Black", "Blue"].

Now, `i` is incremented to 2 for the next iteration, which checks "Blue" instead of the second "Black".

The second "Black" is skipped.

To correct the code, one approach is to decrement `i` when an item is removed, to account for the left shift of elements:

```
```java
static void delAll(List<String> l, String target) {
    for (int i = 0; i < l.size(); i++) {
        if (target.equals(l.get(i))) {
            l.remove(i);
            i--; // Decrement the index to account for the removed item.
        }
    }
}
```
```

Another safer approach is to loop backwards from the end of the list to the beginning. This way, the removal of an element doesn't affect the indexes of the elements that have already been checked:

```
```java
static void delAll(List<String> l, String target) {
    for (int i = l.size() - 1; i >= 0; i--) {
        if (target.equals(l.get(i))) {
            l.remove(i);
        }
    }
}
```
```

Either solution will fix the `IndexOutOfBoundsException` and correctly remove all instances of the "Black" string from the list.

### DA-LE Prompt #1

The code below contains logical errors.

Expected Output: [1, 2, 3, 4, 0, 5, 6]

Actual Output: [1, 2, 3, 4, 5, 6, 0, 0, 0, 0]

Explain the errors and correct the code.

// Code containing logical error

```
import java.util.Arrays;
```

```
public class Debugging {
```

```
 static int[] resize(int[] input, int newSize) {
 return Arrays.copyOf(input, newSize < input.length ? newSize : input.length);
 }
```

```
 static int add(int[] array, int data, int index) {
 for (int i = 0; i <= index; i++) {
```

```

 if (array[i] == data) {
 return index;
 }
 }
 array[index++] = data;
 return index;
}

static int[] generateSet(int... array) {
 int[] set = new int[array.length];
 int idx = 0;
 for (int element : array) {
 idx = add(set, element, idx);
 }
 resize(set, idx);
 return set;
}

static int[] concat(int[] array1, int[] array2) {
 int[] rslt = new int[array1.length + array2.length];
 System.arraycopy(array1, 0, rslt, 0, array1.length);
 System.arraycopy(array2, 0, rslt, array1.length, array2.length);
 return generateSet(rslt);
}

public static void main(String[] args) {
 int[] set1 = generateSet(1, 2, 3, 4, 0),
 set2 = generateSet(0, 3, 4, 5, 6);
 int[] union = concat(set1, set2);
 System.out.println(Arrays.toString(union));
}
}

```

### Response

The logical error in the code arises from the way the program is intended to create a unique set and then concatenate two such sets. Here's the sequence of events causing the error:

1. The method `generateSet()` creates an array called `set` and uses the `add()` function to add elements to it.
2. Inside the `add()` method, if the element is found in the array before the index, the function does not add it again. This works fine if the array does not contain any zeros. However, when you first create the `set`, you initialize it with zeros, and then zeros are treated as valid data.
3. As you are passing arrays with zeros to `concat()`, they are combined and passed to `generateSet()` resulting in a non-unique set of elements.

To fix this problem, you need to differentiate between initialized zeros in the `set` and actual zero values you want to add. Here's the corrected code:

```

```java
import java.util.Arrays;

public class Debugging {

    static int[] resize(int[] input, int newSize) {
        return Arrays.copyOf(input, newSize);
    }
}

```

```

static int add(int[] array, int data, int index) {
    for (int i = 0; i < index; i++) {
        if (array[i] == data) {
            return index;
        }
    }
    array[index] = data;
    return index + 1;
}

static int[] generateSet(int... array) {
    int[] set = new int[array.length];
    int idx = 0;
    for (int element : array) {
        idx = add(set, element, idx);
    }
    return resize(set, idx);
}

static int[] concat(int[] array1, int[] array2) {
    int[] rslt = new int[array1.length + array2.length];
    System.arraycopy(array1, 0, rslt, 0, array1.length);
    System.arraycopy(array2, 0, rslt, array1.length, array2.length);
    return generateSet(rslt);
}

public static void main(String[] args) {
    int[] set1 = generateSet(1, 2, 3, 4, 0),
        set2 = generateSet(0, 3, 4, 5, 6);
    int[] union = concat(set1, set2);
    System.out.println(Arrays.toString(union));
}
...

```

Changes made:

1. Modified the `add()` method:
 - Changed the loop's condition to `i < index` because we only need to check elements up to the `index`.
 - Return the updated index (`index + 1`), after adding the element.
2. In the `resize()` method, the new size of the array should always be `newSize`, so we directly return `Arrays.copyOf(input, newSize)`.

With these changes, the program will output the expected result: `[1, 2, 3, 4, 0, 5, 6]`.