**Q 3: Why is type-checking the parameters of a subprogram important?**

Type-checking refers to verifying that the types of variables or parameters used in a program match the expected types specified by the program’s declarations. In the context of subprograms (functions, procedures, methods), type-checking the parameters is crucial for several reasons:

1. Ensures Correct Usage of Functions/ Methods:
   * When a subprogram is defined, it expects parameters of certain types. Type checking ensures that when the subprogram is called, the arguments passed to it are of the correct types.
   * If parameters are not type-checked, there’s a risk of passing incompatible data types, leading to unexpected behavior, runtime errors, or crashes.
2. Prevents Errors Early:
   * Static type checking (done at compile time in statically – typed languages like C++ and Java) helps catch type-related errors before the program is executed. This is beneficial because it reduces the number of runtime errors, which are harder to debug.
   * For example, passing a string where an integer is expected could result in incorrect calculations or data corruption if not checked properly.
3. Improves Code Reliability and Maintainability:
   * Strong type checking improves the reliability of the code, making it less prone to bugs and easier to maintain. It also helps developers understand the expected behavior of functions or methods, which is important for collaboration of code documentation.
4. Supports Overloading and Polymorphism:
   * In languages that support function/ method overloading or polymorphism, type checking allows the compiler or runtime environment to determine which version of a function to invoke based on the types of the parameters.
5. #include<iostream>
6. using *namespace* std;
7. *void* add(*int* *a*, *int* *b*)
8. {
9. cout << *a* + *b* << endl;
10. }
11. *int* main()
12. {
13. add(5, 10); //Correct, both parameters are integers
14. add("5", 10); // Compilation error, "5" is a string, not an integer
15. }

Here, the compiler would give a compilation error like the one shown in the picture below.

A screen shot of a computer program

Description automatically generated

**Q4: What is aliasing?**

Aliasing occurs when two or more different identifiers (variables, pointers, reference, etc.) refer to the same memory location. In other words, aliasing happens when two or more variables point to the same data in memory. This can occur unintentionally or be used deliberately in programming.

A black and white text

Description automatically generated

Here, y is an alias for x. Modifying y also modifies x because they refer to the same memory location.

Importance:

* **Memory Efficiency**: Aliasing can be useful when you want multiple references to the same data without duplicating the memory used.
* **Potential Pitfalls**: However, aliasing can make debugging difficult. When two variables modify the same memory location, it might not be clear which variable caused a change, leading to subtle bugs.
* **Impact on Optimization**: Aliasing can also hinder compiler optimizations. If the compiler cannot determine whether two variables alias the same memory, it might be forced to make conservative decisions, resulting in less optimized code.

A screenshot of a computer

Description automatically generated

**Q5: What is exception handling?**

Exception handling is a mechanism that allows a program to deal with unexpected events or errors during execution in a controlled and graceful manner. These events are known as exceptions. When an error occurs, an exception is thrown (or raised), and the program can handle it using special code, typically called exception handlers.

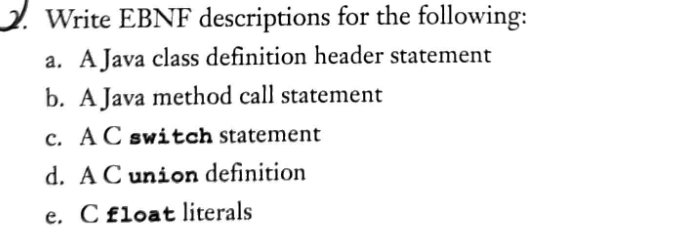
Without exception handling, programs would terminate abruptly when an error occurs. Exception handling allows a program to continue execution even when an error occurs or to clean up resources before terminating gracefully.

Key Concepts:

* Throwing an Exception: When an error occurs, an exception is “thrown” (raised). This could be due to various reasons, such as:
  + Division by zero
  + File not found
  + Null pointer dereferencing
  + Array index out of bounds
* Catching an Exception: Once an exception is thrown, it can be “caught” by an exception handler, which is responsible for taking appropriate action, such as logging the error, cleaning up resources, or retrying the operation.
* Propagation: If an exception is not caught in the function where it occurred, it is propagated up the call stack to higher levels of the program. If the exception is never caught, the program may terminate with an error.

A screen shot of a computer program

Description automatically generated



**2. Write EBNF descriptions for the following**

**a. A Java class definition header statement**

The class header in Java specifies the class’s name, access modifiers, and possibly the inheritance details (e.g., which superclass it extends, which interfaces it implements).

A black screen with colorful text

Description automatically generated



1. Basic Class:

A screen shot of a computer

Description automatically generated

1. Class with Inheritance:

A black screen with white text

Description automatically generated

1. Class with Interface:

A black screen with blue and white text

Description automatically generated

1. Class with Inheritance and Interface:

A black screen with white text

Description automatically generated

**b. A Java Method Call Statement**

A screen shot of a computer

Description automatically generated



Examples and Explanation

Example 1: Simple method Call with No arguments



* Explanation:
  + **printHello** is the **<method-name>**, which is an identifier representing the method.
  + The parenthesis **()** are required, even though there are no arguments.
  + This method doesn’t take any arguments, so the argument-list is empty

Example 2: Method Call with One Argument



* Explanation:
  + PrintMessage is the <method-name>
  + The argument “Hello, World” is a **literal**, which is part of the <argument-list>.
  + The method call is passing one argument to the method, which is a string literal.

This matches the rule

A black screen with white text

Description automatically generated

Here, argument-list consists of one literal expression.

Example 3: Method Call with Multiple Arguments



* Explanation:
  + addNumbers is the <method-name>.
  + 5 and 10 are literals, which are part of the <argument-list>
  + The method call passes two arguments, 5 and 10, which are separated by a comma.

This matches the rule:

A black background with white text

Description automatically generated

The argument-list consists of two literal expressions, separated by a comma.

Example 4: Method Call as an Argument to Another Method

calculate(add(3,4), subtract(10,5));

**Explanation**:

* calculate is the outer <method-call>.
* The arguments passed to calculate are themselves method calls: add(3, 4) and subtract(10, 5).
* add and subtract are method names, and each method takes two literals as arguments.
* This is an example of **nested method calls**.

This matches the rule:

A black background with white text

Description automatically generated

In this case, the argument-list consists of two expressions, and each expression is itself a method call.

c. C switch Statement

// C switch Statement

<switch-statement> ::= "switch" "(" <expression> ")" "{" { <case> } [ <default> ] "}"

<case> ::= "case" <constant-expression> ":" {<statement>}

<default> ::= "default:" {<statement>}

<expression> ::= <identifier> | <literal> | <operation>

<constant-expression> ::= <literal> | <constant-identifier>

Example:

#include<stdio.h>

*int* main(){

    switch (x) {

        case 1:

            doSomething();

            break;

        default:

            doDefault();

    }

}

d. C union Definition:

// C union Definition

<union-definition> ::= "union" <identifier> "{" {<union-member>} "}" [<variable-name>]

<union-member> ::= <type> <identifier> ";"

<type> ::= "int" | "float" | "char" | <custom-type> | ...

Example:

*union* Data {

*int* i;

*float* f;

*char* c;

};