SURVEY INSTRUCTION (1/3)
HOW TO EVALUATE ACCURACY

1. Definition of Accuracy

Accuracy refers to how correctly the generated summary reflects the actual functionality and behaviors of the source code.
This means the summary should not contain false or misleading statements about what the code does.

2. Java Code Sample

```
boolean goFromRetiredToDead() {
    return compareAndSet(Status.RETIRED, Status.DEAD);
}
```

Above is the Java method we're summarizing. It attempts to transition an entry from a `RETIRED` status to a `DEAD` status, returning a boolean indicating success or failure.

3. Example Summaries: Accurate vs. Inaccurate

Below are two example summaries—one accurate, one inaccurate—illustrating the difference in how they describe the same code snippet.

Accurate	Inaccurate
Summary 1 Attempts to transition the entry from retired to dead when releasing the handle.	**Summary 2** Attempts to transition the entry from idle to customise when evicting from the idle cache.
Why is this accurate? It correctly states that the code tries to change the status from `RETIRED` to `DEAD`. There are no misleading or false claims; it matches the logic of `compareAndSet(Status.RETIRED, Status.DEAD)`.	**Why is this inaccurate?** It mentions switching from `idle` to `customise`, which is not present in the code. It also references an "idle cache," which is nowhere in the method, making this summary misleading and factually incorrect.

4. How to Apply This Example in Your Evaluation

- **Compare the Summary to the Actual Code**
  - Check whether the status, variables, or logic in the summary accurately match what the code does.
- **Rank the Summary by Accuracy**
  - A summary that cleanly aligns with the code's functionality earns a higher (better) rank.
  - In this example, **Summary 1** is ranked 1 (more accurate, higher rank), while **Summary 2** is ranked number 2 (less accurate, lower rank).
- **Use this approach for each code snippet and summary pair in your survey, assigning ranks that reflect how accurately each summary describes the code.**

SURVEY INSTRUCTION (2/3)
HOW TO EVALUATE CONTENT ADEQUACY

1. Definition of Content Adequacy

Content Adequacy refers to the amount of important information about the source code successfully captured and conveyed in the summary.
A summary with good content adequacy should include all necessary details for understanding the code's purpose and behavior, without omitting key points.

2. Java Code Sample

```
private Tag clone(final ITreeNode<CTag> currentNode, final Tag parentExpression) {
    final Tag childExpression = new Tag(currentNode);
    m_allTags.put(currentNode, childExpression);

    if (parentExpression != null) {
        Tag.link(parentExpression, childExpression);
    }

    for (final ITreeNode<CTag> child : currentNode.getChildren()) {
        clone(child, childExpression);
    }

    return childExpression;
}
```

This code recursively clones a tag tree, linking each new `Tag` object to its parent, and stores it in a mapping (`m_allTags`). It creates a `childExpression` for each node and returns the newly cloned tree.

3. Example Summaries: Adequate vs. Inadequate

Below are two example summaries—one demonstrating higher content adequacy, the other demonstrating lower content adequacy.

Adequate	Inadequate
Summary 1 Converts an internal tag tree into an API tag tree.	**Summary 2** Creates a new empty tree node.
Why is this more adequate? It captures the essence of the operation—building a new structure (API tag tree) from an existing one—thereby reflecting the core cloning and linking logic. While it doesn't mention every detail (like recursion or the mapping), it conveys the most important aspect of the code.	**Why is this inadequate?** Although this summary mentions the creation of a tree node, it inaccurately describes it as "empty" and does not capture the recursive cloning process or the linking of the new node with its parent. It only partially touches on the operation (node creation) without the broader context.

4. How to Apply This Example in Your Evaluation

- **Compare the Summary to the Actual Code**
  - Check whether the summary includes crucial details—such as recursively creating new tags, linking them to the parent, and using a mapping (`m_allTags`).
- **Rank the Summary by Content Adequacy**
  - A summary covering the essential logic (cloning, linking, recursion) deserves a higher rank.
  - In this example, **Summary 1** is rank 1 (more adequate), and **Summary 2** is rank 2 (less adequate).
- **Use This Approach for Each Snippet and Summary**
  - When ranking, focus on how thoroughly the summary conveys the code's purpose, behavior, and structure.
  - Summaries that omit important details or stray from what the code actually does earn a lower rank for content adequacy.

SURVEY INSTRUCTION (3/3)
HOW TO EVALUATE CONCISENESS

1. Definition of Conciseness

Conciseness refers to how succinctly the summary conveys only the essential information needed to understand the code, avoiding superfluous words or irrelevant details.
A concise summary should capture all key points without unnecessary wording, maintaining clarity while minimizing excess.

2. Java Code Sample

```java
private static MethodHandle selectNumberTransformer(Class param, Object arg) {
    param = TypeHelper.getWrapperClass(param);
    if (param == Byte.class) {
        return TO_BYTE;
    } else if (param == Character.class || param == Integer.class) {
        return TO_INT;
    } else if (param == Long.class) {
        return TO_LONG;
    } else if (param == Float.class) {
        return TO_FLOAT;
    } else if (param == Double.class) {
        return TO_DOUBLE;
    } else if (param == BigInteger.class) {
        return TO_BIG_INT;
    } else if (param == BigDecimal.class) {
        return TO_BIG_DEC;
    } else if (param == Short.class) {
        return TO_SHORT;
    } else {
        return null;
    }
}
```

This code checks a given parameter's type, converts it to a wrapper class if needed, and returns a suitable `MethodHandle` for transforming one number type into another. If no matching transformer is found, it returns `null`.

3. Example Summaries: Concise vs. Less Concise

Below are two example summaries—one more concise, one less concise—illustrating how they present the same code snippet.

Concise	Less Concise
Summary 1 Helper method to check if the given type is a valid value.	**Summary 2** Returns a transformer later applied as a filter to transform one number into another, also depending on whether the parameter matches recognized numeric wrapper classes.
Why is this more concise? It delivers its core message with minimal words, focusing on the code's purpose—verifying a type and returning a handle—without additional fluff.	**Why is this less concise?** It adds extra context about filters and recognized numeric classes, making the statement longer. While informative, it's more verbose than necessary to convey the primary function.

4. How to Apply This Example in Your Evaluation

- **Check for Unnecessary Details**
  - See if the summary stays focused on the essential function or includes extra wording that doesn't enhance understanding.
- **Rank the Summary by Conciseness**
  - A more concise summary that still conveys the key idea earns a higher rank.
  - Here, **Summary 1** should be ranked 1 (higher concise), while **Summary 2** is ranked 2 (less concise).
- **Use This Approach for Each Snippet and Summary**
  - When ranking conciseness, consider how succinctly the summary captures the code's purpose without adding unnecessary length.
  - Ensure essential details are not omitted for the sake of brevity.