Solution to Problem 3.4.5

Problem Is the following implementation of `hashCode()` legal?

```
1 public int hashCode() {
2     return 17;
3 }
```

—

Is it Legal?

Yes, this implementation of `hashCode()` is **legal** because:

1. **Legal Definition**: In Java, the `hashCode()` method is expected to return an integer value. The method signature and return type are correct.
2. **Contract with `Object`**: The implementation satisfies the `hashCode()` method required by the `Object` class.

Thus, the implementation is syntactically and functionally valid.

—

Effect of Using This Implementation

While the implementation is legal, it is highly **inefficient** and violates the intended purpose of `hashCode()` because:

1. **Hash Collisions**:
   - Since the `hashCode()` method always returns the same value (`17`), all objects of this class will have the same hash code.
   - This causes **all objects to be stored in the same bucket** in hash-based collections like `HashMap` or `HashSet`.
2. **Performance Degradation**:
   - Hash-based collections rely on the hash code to distribute objects across buckets evenly.
   - With all objects mapped to the same bucket, these collections will degrade from $O(1)$ (amortized) to $O(n)$ for operations like `get()`, `put()`, or `contains()`.
3. **Equality Consistency**:
   - While this implementation does not violate the `equals-hashCode` contract, it is not meaningful for practical use cases.
   - Objects that are not equal (`!obj1.equals(obj2)`) might still have the same hash code due to the constant return value, which is undesirable in most cases.

—

When Would This Implementation Be Useful?

This implementation might be acceptable in the following scenarios:

- If the object is not intended to be stored in hash-based collections (e.g., `HashMap`, `HashSet`).
- If the object is always compared using the `equals()` method and the hash code is irrelevant.
- In testing environments, where a simple hash code is sufficient for demonstration purposes.

—

Example of Effect in a HashMap

The following example demonstrates the inefficiency of using a constant `hashCode()`.

```java
import java.util.HashMap;

public class Example {
    public static void main(String[] args) {
        HashMap<MyClass, String> map = new HashMap<>();
        map.put(new MyClass(), "Value1");
        map.put(new MyClass(), "Value2");
        System.out.println("Size of map: " + map.size());
    }
}

class MyClass {
    @Override
    public int hashCode() {
        return 17; // Constant hash code
    }

    @Override
    public boolean equals(Object obj) {
        return this == obj; // Default equality
    }
}
```

Output

```
Size of map: 2
```

Explanation

- Despite all objects having the same hash code (`17`), the `equals()` method differentiates between the objects, so both are stored in the `HashMap`.

- However, performance suffers because all objects are stored in the same bucket.

—

Summary

- **Legality**: The implementation is legal but suboptimal.
- **Effect**: Causes severe hash collisions, leading to performance degradation in hash-based collections.
- **Use Case**: Rarely useful in practice unless hash code is irrelevant or for testing/demonstration purposes.

The correct implementation of `hashCode()` should distribute objects more uniformly across hash buckets to optimize performance.