EXP55-J. Use the same type for the second and third operands in conditional expressions

The conditional operator `?:` uses the `boolean` value of its first operand to decide which of the other two expressions will be evaluated. (See §15.25, "Conditional Operator ? :" of the *Java Language Specification* (JLS) [JLS 2013].)

The general form of a Java conditional expression is `operand1 ? operand2 : operand3`.

- If the value of the first operand (`operand1`) is `true`, then the second operand expression (`operand2`) is chosen.
- If the value of the first operand is `false`, then the third operand expression (`operand3`) is chosen.

The conditional operator is syntactically right-associative. For example, `a?b:c?d:e?f:g` is equivalent to `a?b:(c?d:(e?f:g))`.

The JLS rules for determining the result type of a conditional expression (see following table) are complicated; programmers could be surprised by the type conversions required for expressions they have written.

Result type determination begins from the top of the table; the compiler applies the first matching rule. The Operand 2 and Operand 3 columns refer to `operand2` and `operand3` (from the previous definition) respectively. In the table, `constant int` refers to constant expressions of type `int` (such as `'0'` or variables declared `final`).

For the final table row, S1 and S2 are the types of the second and third operands respectively. T1 is the type that results from applying boxing conversion to S1, and T2 is the type that results from applying boxing conversion to S2. The type of the conditional expression is the result of applying capture conversion to S2. The type of the conditional expression is the result of applying capture conversion to the least upper bound of T1 and T2. See §5.1.7, "Boxing Conversion," §5.1.10, "Capture Conversion," and §15.12.2.7, "Inferring Type Arguments Based on Actual Arguments," of the JLS for additional information [JLS 2013].

Determining the Result Type of a Conditional Expression

Rule	Operand 2	Operand 3	Resultant Type
1	Type T	Type T	Type T
2	`boolean`	`Boolean`	`boolean`
3	`Boolean`	`boolean`	`boolean`
4	`null`	`reference`	`reference`
5	`reference`	`null`	`reference`
6	`byte` or `Byte`	`short` or `Short`	`short`
7	`short` or `Short`	`byte` or `Byte`	`short`
8	`byte`, `short`, `char`, `Byte`, `Short`, `Character`	`constant int`	`byte`, `short`, `char` if value of `int` is representable
9	`constant int`	`byte`, `short`, `char`, `Byte`, `Short`, `Character`	`byte`, `short`, `char` if value of `int` is representable
10	Other numeric	Other numeric	Promoted type of the second and third operands
11	T1 = boxing conversion(S1)	T2 = boxing conversion(S2)	Apply capture conversion to lub(T1,T2)

The complexity of the rules that determine the result type of a conditional expression can lead to unintended type conversions. Consequently, the second and third operands of each conditional expression should have identical types. This recommendation also applies to boxed primitives.

Noncompliant Code Example

In this noncompliant code example, the programmer expects that both print statements will print the value of `alpha` as a `char`:

```
public class Expr {
  public static void main(String[] args) {
    char alpha = 'A';
    int i = 0;
    // Other code. Value of i may change
    boolean trueExp = true; // Some expression that evaluates to true
    System.out.print(trueExp ? alpha : 0); // prints A
    System.out.print(trueExp ? alpha : i); // prints 65
  }
}
```

The first print statement prints `A` because the compiler applies rule 8 from the result type determination table to determine that the second and third operands of the conditional expression are, or are converted to, type `char`. However, the second print statement prints `65`—the value of `alpha` as an `int`. The first matching rule from the table is rule 10. Consequently, the compiler promotes the value of `alpha` to type `int`.

Compliant Solution

This compliant solution uses identical types for the second and third operands of each conditional expression; the explicit casts specify the type expected by the programmer:

```
public class Expr {
  public static void main(String[] args) {
    char alpha = 'A';
    int i = 0;
    boolean trueExp = true; // Expression that evaluates to true
    System.out.print(trueExp ? alpha : ((char) 0)); // Prints A
    // Deliberate narrowing cast of i; possible truncation OK
    System.out.print(trueExp ? alpha : ((char) i)); // Prints A
  }
}
```

When the value of `i` in the second conditional expression falls outside the range that can be represented as a `char`, the explicit cast will truncate its value. This usage complies with exception **NUM12-J-EX0** of NUM12-J. Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data.

Noncompliant Code Example

This noncompliant code example prints 100 as the size of the `HashSet` rather than the expected result (some value between 0 and 50):

```
public class ShortSet {
  public static void main(String[] args) {
    HashSet<Short> s = new HashSet<Short>();
    for (short i = 0; i < 100; i++) {
      s.add(i);
      // Cast of i-1 is safe because value is always representable
      Short workingVal = (short) (i-1);
      // ... Other code may update workingVal

      s.remove(((i % 2) == 1) ? i-1 : workingVal);
    }
    System.out.println(s.size());
  }
}
```

The combination of values of types `short` and `int` in the second argument of the conditional expression (the operation `i-1`) causes the result to be an `int`, as specified by the integer promotion rules. Consequently, the `Short` object in the third argument is unboxed into a `short`, which is then promoted into an `int`. The result of the conditional expression is then autoboxed into an object of type `Integer`. Because the `HashSet` contains only values of type `Short`, the call to `HashSet.remove()` has no effect.

Compliant Solution

This compliant solution casts the second operand to type `short`, then explicitly invokes the `Short.valueOf()` method to create a `Short` instance whose value is `i-1`:

```
public class ShortSet {
  public static void main(String[] args) {
    HashSet<Short> s = new HashSet<Short>();
    for (short i = 0; i < 100; i++) {
      s.add(i);
      // Cast of i-1 is safe because the resulting value is always representable
      Short workingVal = (short) (i-1);
      // ... Other code may update workingVal

      // Cast of i-1 is safe because the resulting value is always representable
      s.remove(((i % 2) == 1) ? Short.valueOf((short) (i-1)) : workingVal);
    }
    System.out.println(s.size());
  }
}
```

As a result of the cast, the second and third operands of the conditional expression both have type `Short`, and the `remove()` call has the expected result.

Writing the conditional expression as `((i % 2) == 1) ? (short) (i-1)) : workingVal` also complies with this guideline because both the second and third operands in this form have type `short`. However, this alternative is less efficient because it forces unboxing of `workingVal` on each even iteration of the loop and autoboxing of the result of the conditional expression (from `short` to `Short`) on every iteration of the loop.

Applicability

When the second and third operands of a conditional expression have different types, they can be subject to unexpected type conversions.

Automated detection of condition expressions whose second and third operands are of different types is straightforward.

Bibliography

[Bloch 2005]	Puzzle 8, "Dos Equis"
[Findbugs 2008]	"Bx: Primitive Value Is Unboxed and Coerced for Ternary Operator"
[JLS 2013]	§15.25, "Conditional Operator ? :"