1Z0-808 Exam Topic Reviewer

TopicId: 1005
Topic: Type Conversion and Casting

August 5, 2025

Topic 1005: Type Conversion and Casting

Thinking Like the Compiler: Shifting Shapes

You've mastered the primitive types. Now we explore how Java converts data from one type to another. This happens constantly in expressions. The compiler has strict rules for this, and understanding them is key to deconstructing many exam questions. These questions often look like simple arithmetic, but they are secretly testing your knowledge of type promotion and casting.

Implicit Conversion (Widening)

This is an automatic and safe conversion. It happens when you assign a value from a "smaller" data type to a "larger" one. No data is lost in terms of magnitude. **The Widening Path:** `byte` → `short` → `int` → `long` → `float` → `double` A `char` can also be widened to an `int` and follow the path from there.

```
int myInt = 100;
long myLong = myInt; // Implicit conversion from int to long
float myFloat = myLong; // Implicit conversion from long to float
```

Exam Trap: While widening from an integer type (`int`, `long`) to a floating-point type (`float`, `double`) doesn't lose magnitude, it can lose *precision*. A very large `long` may not have an exact representation as a `float`. The exam is unlikely to ask you to calculate this, but you should be aware of the concept.

Explicit Conversion (Narrowing / Casting)

This is a manual conversion from a "larger" type to a "smaller" one. You must explicitly tell the compiler to do it using the cast operator `(type)`. This is risky and can result in data loss.

```
double price = 299.99;
// int roundedPrice = price; // COMPILE ERROR! Possible lossy conversion
int truncatedPrice = (int) price; // OK. truncatedPrice is now 299
```

Key Point: Casting from a floating-point to an integer type **truncates** the value (cuts off the decimal part). It does **not** round.

Overflow in Narrowing

When you cast a value that is too large for the target type, it "overflows" or "wraps around". The result is calculated by taking the remainder of a division by the target type's range.

```
// A byte can hold values from -128 to 127.
int i = 257;
byte b = (byte) i; // 257 is too large for a byte.
// Result is 257 % 256 = 1. So, b will be 1.
System.out.println(b); // Prints 1
```

Arithmetic Promotion Rules: The Core of the Trap

This is one of the most tested concepts in this area. When you perform an arithmetic operation, Java automatically promotes smaller types.

Rule 1: If one operand is a `double`, the other is converted to a `double`.

Rule 2: Otherwise, if one operand is a `float`, the other is converted to a `float`.

Rule 3: Otherwise, if one operand is a `long`, the other is converted to a `long`.

Rule 4: Otherwise, **both operands are converted to an** `int`.

Rule 4 is the source of many errors. It means that any operation on types smaller than `int` (`byte`, `short`, `char`) will result in an `int`.

```
short s1 = 10;
short s2 = 20;
// short s3 = s1 + s2; // COMPILE ERROR!
// Why? s1 and s2 are promoted to int. The result of s1 + s2 is an int.
// You are trying to assign an int to a short, which is a narrowing conversion.

// The fix is to cast:
short s3 = (short) (s1 + s2); // OK
```

Super-Trap: Compound Assignment Operators

The compound assignment operators (`+=, -=, *=, /=, %=`) have a hidden, built-in cast. This is a deliberate "gotcha" on the exam.

Let's revisit the previous example:

```
short s1 = 10;
short s2 = 20;

// s1 = s1 + s2; // Still a COMPILE ERROR.

s1 += s2; // THIS WORKS!
// The line above is equivalent to: s1 = (short) (s1 + s2);
// The compound operator automatically casts the result back to the type
// of the variable on the left.
```

A Final Nuance: Compile-Time Constants

If you assign a literal or a `final` variable to a smaller type, the compiler is smart. If it can determine at compile time that the value fits, it will allow the assignment without an explicit cast.

```
// Example 1: Literals
byte b1 = 100; // OK. 100 fits in a byte.
// byte b2 = 200; // COMPILE ERROR. 200 is too large for a byte.

// Example 2: final variables
```

```
final int i = 50;
byte b3 = i; // OK. Compiler knows i is 50 and it fits.

int j = 50;
// byte b4 = j; // COMPILE ERROR. 'j' is not final, its value could change.
// So the compiler insists on a cast: byte b4 = (byte) j;
```

Key Takeaways for the 1Z0-808 Exam

- **Casting Truncates:** Remember `(int) 99.99` is `99`.
- **Promotion to `int`:** Any arithmetic with `byte`, `short`, or `char` results in an `int`.
- **Compound Operators Cast:** `x += y` is not the same as `x = x + y`. The first one has an implicit cast.
- **Literals and `final`:** The compiler is lenient with assignments from literals and `final` variables if the value is known to fit.