Java 21

Type Conversions

A Complete Beginner's Tutorial

■ Welcome! This tutorial assumes you are brand new to Java. Every concept is explained in plain English before any code is shown. Work through the sections in order and you will have a solid understanding of how Java moves data from one type to another.

1. What is a Type?

In Java every piece of data has a **type** — a label that tells the computer what kind of value it is and how much memory to use for it. For example, the number **42** is an *integer*, the number **3.14** is a *decimal*, and the word **"hello"** is a *String*.

Java has two broad families of types:

Family	Examples	What they store
Primitive types	int, double, boolean, char …	Simple raw values
Reference types	String, Integer, Double …	Objects (richer data + methods)

A **type conversion** (also called a *type cast*) is simply asking Java to treat a value as a different type. Sometimes Java does this automatically; other times you must tell it explicitly.

2. Primitive Type Conversions

Java's primitive types are ordered by size. Think of them like containers — a small value always fits into a bigger container, but pouring a big container into a small one may cause **overflow** (data loss).

2.1 Widening — Automatic (Implicit) Conversion

When you put a smaller type into a larger type Java does the conversion **automatically** — no extra code needed. This is called a *widening conversion* and is always safe.

■ Size order (smallest → largest): byte (8-bit) → short (16-bit) → int (32-bit) → long (64-bit) → float → double (64-bit)

```
// Widening – Java does this automatically
byte  b = 42;
short s = b;       // byte  → short   ✓
int   i = s;       // short → int     ✓
long  l = i;       // int   → long    ✓
float f = l;       // long  → float   ✓
double d = f;      // float → double  ✓

System.out.println(d);  // 42.0
```

2.2 Narrowing — Manual (Explicit) Cast

Going the other direction — large type into a small one — requires you to write a **cast** using parentheses. Java will not do this automatically because data loss is possible.

```
// Narrowing – you must write the cast
double d = 9.99;
int    i = (int) d;   // Cast: double → int

System.out.println(i);  // 9  ← decimal part is LOST, not rounded
```

■■ Warning: narrowing truncates, it does not round. 9.99 becomes 9, not 10. Always double-check that the value fits in the target type.

2.3 The Special Case of char

The **char** type stores a single character using its Unicode number (e.g. 'A' = 65). It is 16-bit unsigned, which creates a few surprises:

```
char  c = 'A';
int   i = c;          // Widening: char → int  (i = 65)

int   n = 66;
```

```
char  ch = (char) n;  // Narrowing: int → char  (ch = 'B')

// byte → char is also a narrowing cast (byte is signed, char is unsigned)
byte  by = 65;
char  ch2 = (char) by;  // ch2 = 'A'
```

3. Boxing and Unboxing

Primitives are fast and lightweight, but sometimes you need an **object** — for example, to store numbers in a List. Java provides a **wrapper class** for every primitive type. Converting between them is called *boxing* and *unboxing*.

Primitive	Wrapper class
int	Integer
long	Long
double	Double
float	Float
boolean	Boolean
char	Character
byte	Byte
short	Short

Autoboxing and Autounboxing

Since Java 5, the compiler handles boxing and unboxing **automatically** in most situations — you rarely need to write it yourself.

```
// Autoboxing: primitive → wrapper (automatic)
int     primitive = 42;
Integer boxed     = primitive;   // Java boxes it for you

// Autounboxing: wrapper → primitive (automatic)
Integer wrapped = 100;
int     raw     = wrapped;       // Java unboxes it for you

// Common use-case: storing in a List (requires objects, not primitives)
java.util.List<Integer> list = new java.util.ArrayList<>();
list.add(7);    // 7 is autoboxed to Integer(7)
int val = list.get(0);  // autounboxed back to int
```

■■ Null danger: if a wrapper object is null and Java tries to autounbox it, you get a NullPointerException at runtime. Always check for null before unboxing a variable that could be null.

4. Reference Type Conversions

Java classes are organised in a hierarchy. Every class has a parent (superclass), and at the very top sits **Object**. Moving up the hierarchy is automatic; moving down requires a cast.

4.1 Upcasting — Automatic

Assigning a more specific type to a more general variable is always safe and automatic. This is called *upcasting*.

```
// String is a subclass of Object
String  str = "Hello, Java!";
Object  obj = str;          // Upcasting — automatic, always safe

// Dog extends Animal
Animal animal = new Dog();  // Upcasting — automatic
```

4.2 Downcasting — Manual

Going the other way — treating a general type as a specific one — requires an explicit cast. If the actual object is not of that type, Java throws a **ClassCastException** at runtime.

```
Object obj = "Hello";         // obj holds a String underneath

// Downcast: Object → String
String str = (String) obj;    // Works fine — it really IS a String

// Dangerous downcast:
Object num = Integer.valueOf(42);
String bad = (String) num;    // ■ ClassCastException at runtime!
```

4.3 Safe Downcasting with instanceof (Java 16+)

Use the **instanceof** keyword to check the type before casting. Java 16+ introduced *pattern matching* which does the check and the cast in one step.

```
Object obj = "Hello, World!";

// Classic style (pre Java 16)
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}

// Pattern matching style (Java 16+, recommended)
if (obj instanceof String s) {
    System.out.println(s.length());  // s is already a String here
}
```

4.4 Pattern Matching in Switch (Java 21)

Java 21 made *pattern matching in switch* a permanent feature, letting you cleanly handle multiple types in one block:

```
Object value = 3.14;

String description = switch (value) {
    case Integer i -> "Integer: " + i;
    case Double  d -> "Double: "  + d;
    case String  s -> "String: "  + s;
    default        -> "Unknown type";
};

System.out.println(description);  // Double: 3.14
```

5. String Conversions

Converting to and from **String** is one of the most common tasks in Java. Unlike primitive conversions, these use method calls rather than casts.

5.1 Any Type → String

```
int    i = 42;
double d = 3.14;
boolean b = true;

// Option A: String.valueOf() — recommended, null-safe
String s1 = String.valueOf(i);   // "42"
String s2 = String.valueOf(d);   // "3.14"
String s3 = String.valueOf(b);   // "true"

// Option B: concatenation with ""
String s4 = i + "";              // "42"

// Option C: wrapper .toString()
String s5 = Integer.toString(i); // "42"

// Objects use their .toString() method
Object obj = new java.util.ArrayList<>();
String s6 = obj.toString();      // "[]"
```

5.2 String → Number

```
String numStr = "42";
String decStr = "3.14";

// Parse to primitive
int    i = Integer.parseInt(numStr);    // 42
double d = Double.parseDouble(decStr);  // 3.14
long   l = Long.parseLong("9999999");   // 9999999

// Parse to wrapper object
Integer obj = Integer.valueOf(numStr);  // Integer(42)

// ■■ Parsing a non-numeric string throws NumberFormatException
try {
    int bad = Integer.parseInt("hello"); // ■
} catch (NumberFormatException e) {
    System.out.println("Not a valid number!");
}
```

5.3 The String ↔ double Round Trip

Converting a String to a double and back again *usually* preserves the value, but the exact string representation may differ. Here is a full worked example:

```
public class RoundTripDemo {
    public static void main(String[] args) {

        // ■■ Normal case ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        String s1   = "3.14159";
        double d1   = Double.parseDouble(s1);
        String back1 = String.valueOf(d1);
        System.out.println(s1 + " → " + back1 + "  match=" + s1.equals(back1));
        // 3.14159 → 3.14159  match=true

        // ■■ Trailing zero lost ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        String s2   = "3.10";
        double d2   = Double.parseDouble(s2);
        String back2 = String.valueOf(d2);
        System.out.println(s2 + " → " + back2 + "  match=" + s2.equals(back2));
        // 3.10 → 3.1  match=false

        // ■■ Scientific notation ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        String s3   = "0.000001";
        double d3   = Double.parseDouble(s3);
        String back3 = String.valueOf(d3);
        System.out.println(s3 + " → " + back3 + "  match=" + s3.equals(back3));
        // 0.000001 → 1.0E-6  match=false

        // ■■ Precision loss ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        String s4   = "1.12345678901234567890";
        double d4   = Double.parseDouble(s4);
        String back4 = String.valueOf(d4);
        System.out.println(s4 + " → " + back4 + "  match=" + s4.equals(back4));
        // 1.12345678901234567890 → 1.1234567890123457  match=false
    }
}
```

5.4 Using BigDecimal for Exact Precision

When the exact decimal representation matters — for money, measurements, or user-visible output — use **java.math.BigDecimal** instead of double.

```
import java.math.BigDecimal;

public class BigDecimalDemo {
    public static void main(String[] args) {

        String original = "3.10";

        // double loses the trailing zero
        double d = Double.parseDouble(original);
        System.out.println(String.valueOf(d));       // 3.1  ← trailing 0 gone

        // BigDecimal preserves the exact representation
        BigDecimal bd = new BigDecimal(original);
        System.out.println(bd.toPlainString());       // 3.10 ✓

        // Arithmetic with BigDecimal is also exact
        BigDecimal price = new BigDecimal("19.99");
        BigDecimal tax   = new BigDecimal("0.08");
        BigDecimal total = price.multiply(tax);
        System.out.println(total.toPlainString());    // 1.5992
    }
}
```

■ Rule of thumb: use double for physics/science calculations where tiny rounding errors don't matter. Use BigDecimal for money or anywhere the exact decimal representation must be preserved.

6. Quick Reference

From	To	How	Safe?
Smaller primitive	Larger primitive	Automatic (widening)	■ Always
Larger primitive	Smaller primitive	Explicit cast (int) x	■■ May lose data
Primitive	Wrapper class	Autoboxing (automatic)	■ Always
Wrapper class	Primitive	Autounboxing (automatic)	■■ Null danger
Subclass	Superclass	Automatic (upcasting)	■ Always
Superclass	Subclass	Explicit cast (Dog) animal	■■ Check with instanceof
Any type	String	String.valueOf(x) or x + ""	■ Always
String	int / long …	Integer.parseInt(s) etc.	■■ May throw exception
String	double	Double.parseDouble(s)	■■ May throw exception
String / double	Exact decimal	new BigDecimal(s)	■ Exact

Key Takeaways

Widening is free. Going from a smaller to a larger primitive type is always automatic and safe.

Narrowing costs you. Going the other way requires a cast and may silently lose data.

Autoboxing is convenient but watch for null. Unboxing a null wrapper throws NullPointerException.

Always check before downcasting. Use instanceof (or pattern matching) before casting a reference type down the hierarchy.

Parse carefully. Integer.parseInt() and Double.parseDouble() throw NumberFormatException on bad input — wrap them in try/catch.

Use BigDecimal for money. double cannot represent most decimal fractions exactly; BigDecimal can.

Happy coding! Type conversion is one of those things that feels confusing at first but quickly becomes second nature once you've written a few Java programs. When in doubt, refer back to the Quick Reference table above.