# Overriding vs Overloading in Java — Quick Notes

**Crisp comparison + tiny examples**

| Aspect | Overriding | Overloading |
|---|---|---|
| Purpose | Change behavior in subclass | Same method name with different parameters |
| Where | Across inheritance (parent ↔ child) | Usually in the same class (or subclass) |
| Signature | Same name & same parameter types/order | Same name, different parameter list |
| Return type | Same or covariant | Can differ but not the only change |
| Binding | Runtime (dynamic dispatch) | Compile-time (method selection) |
| Polymorphism | Yes | No (convenience/compile-time) |
| Modifiers | Cannot override final/static/private | Independent per method |
| Exceptions | Cannot throw broader checked exceptions | Unrelated across overloads |

## Tiny examples

### *Overriding (runtime dispatch)*

```
class A { void f() { System.out.println("A"); } }
class B extends A { @Override void f() { System.out.println("B"); } }

A x = new B();
x.f(); // prints B (runtime dispatch)
```

### *Overloading (compile-time selection)*

```
void sum(int a, int b) {}
void sum(long a, long b) {}      // overload
void sum(int a, int b, int c) {} // overload

// Chosen at compile-time based on argument types
```

## Gotchas

- static methods are hidden, not overridden.
- Overloading + autoboxing/varargs may cause ambiguity—be explicit with types.
- Use @Override to catch mistakes early.

## When to use

- **Override**: customize/extend parent behavior in subclasses.
- **Overload**: offer convenient variants for different parameter shapes.