Interview Cake

# Mutable vs Immutable Objects

A **mutable** object can be changed after it's created, and an **immutable** object can't.

In Java, everything (except for strings) is mutable by default:

Java

```
public class IntegerPair {
    int x;
    int y;

    IntegerPair(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

IntegerPair p = new IntegerPair(5, 10);
// p.x = 5, p.y = 10

p.x = 50;
// p.x = 50, p.y = 10
```

There's no way to make existing objects immutable. Even if an object is declared `final`, its fields can still be changed:

Java

```
public class IntegerPair {
    int x;
    int y;

    IntegerPair(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

final IntegerPair p = new IntegerPair(5, 10);
// p.x = 5, p.y = 10

p.x = 50;
// p.x = 50, p.y = 10
```

That said, if you're defining your own class, you can make its objects immutable by making all fields `final` and `private`.

Java

```
public class IntegerPair {
    private final int x;
    private final int y;

    IntegerPair(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

IntegerPair p = new IntegerPair(5, 10);

p.x = 50;
// Compilation error: cannot assign a value to a final variable
```

Strings can be mutable or immutable depending on the language.

Strings are immutable in Java.

Any time you change a string (e.g.: tacking on an extra character, making it lowercase, swapping two characters), you're actually creating a new and separate copy:

Java

```
String first = "first";

System.out.println(first.hashCode());
// prints something

first = first + "!";

System.out.println(first.hashCode());
// different string, different hash code
```

But in some other languages, like C++, strings can be mutable, and we can modify them directly:

C++

```
string testString("mutable?");

testString[7] = '!';
// testString is now "mutable!"
```

If you want mutable strings in Java, you can use a `StringBuilder` object:

Java

```
StringBuilder mutableString = new StringBuilder("mutable?");

mutableString.setCharAt(7, '!');
// still the same object!
// mutableString is now "mutable!"

// convert to an immutable string
String immutableString = mutableString.toString();
```

Or, you can convert the string to an array of characters, which will be mutable.

Mutable objects are nice because you can make changes **in-place**, without allocating a new object. But be careful—whenever you make an in-place change to an object, *all* references to that object will now reflect the change.



# What's next?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

**Try some questions now ➜**

---

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.