**Kotlin**

In Kotlin, the memory consumed by each data type depends on the underlying JVM (Java Virtual Machine) implementation. Kotlin compiles to JVM bytecode, so its primitive types are mapped to Java's primitives when possible. Here’s a summary of the memory consumption for each data type:

**Data Structures source**

**<https://visualgo.net/en>**

**### Primitive Data Types:**

**1. \*\*Byte\*\***

- Size: 1 byte (8 bits)

- Range: -128 to 127

**2. \*\*Short\*\***

- Size: 2 bytes (16 bits)

- Range: -32,768 to 32,767

**3. \*\*Int\*\***

- Size: 4 bytes (32 bits)

- Range: -2^31 to 2^31 - 1

**4. \*\*Long\*\***

- Size: 8 bytes (64 bits)

- Range: -2^63 to 2^63 - 1

**5. \*\*Float\*\***

- Size: 4 bytes (32 bits)

- Range: Approximately ±3.40282347E+38F

**6. \*\*Double\*\***

- Size: 8 bytes (64 bits)

- Range: Approximately ±1.7976931348623157E+308

**7. \*\*Char\*\***

- Size: 2 bytes (16 bits)

- Unicode character: 0 to 65,535 (unsigned 16-bit integer)

**8. \*\*Boolean\*\***

- Size: 1 bit (internally, JVM may allocate 1 byte or 4 bytes depending on the context)

**### Non-Primitive Data Types (Objects):**

- \*\*String\*\*: The memory usage for a string depends on its length. Each `Char` in a `String` occupies 2 bytes, plus there is overhead for the `String` object itself (around 40 bytes of object overhead on a typical JVM).

**- \*\*Array\*\***: Arrays have overhead, including a header (typically 12 to 16 bytes) plus the space for each element, which depends on the data type.

**### Kotlin-specific types:**

**1. \*\*Nullable Types\*\*:** Nullable types like `Int?`, `Long?`, etc., are boxed types in Kotlin and have extra overhead compared to their primitive counterparts. For example, `Int?` is represented as `Integer` in JVM, which uses 16 bytes or more depending on the object.

**2. \*\*Classes and Objects\*\*:** The memory footprint for user-defined classes and objects depends on the number of properties and their types, including 12–16 bytes of object header plus the size of all the fields (variables) inside the object.

To sum up:

- Primitive types are memory-efficient, with specific fixed sizes.

- Objects and arrays carry additional overhead due to object headers, references, and internal padding.