**Primitives vs Objects (Primitives vs Reference Types)**

How primitives and objects are stored in memory -

**A screenshot of a computer

Description automatically generated with medium confidence**

* **Primitive Example** - Age is preserved as the old value (30) is stored in oldAge before it is re-assigned to 31
* **Object Example –** Friend is a copy of the me object. Then the friend.age is changed from 25 to 50. Yet, when console logging friend.age, the value logged is **still 25**

**Storing in Memory**

**Graphical user interface, diagram

Description automatically generated**

* All **primitive** types are stored in the **call stack**
* All **reference** types are stored in the **HEAP**

**Diagram

Description automatically generatedPrimitives**

* For Primitive types, JavaScript creates a **unique identifier**, then will **allocate a memory address** to **hold the** **value** – Same structure as storing a value in an object – identifier.address.value
* Identifier always points to address first – **Identifier -> Address**
* **Values** at a **certain memory address** are **immutable**, they cannot be changed

**Example**

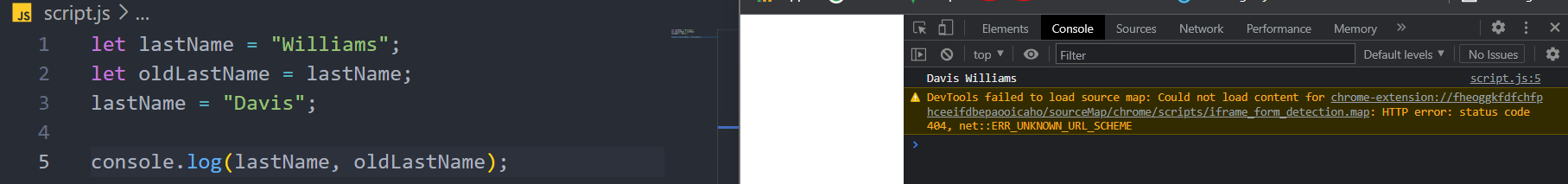
* When **oldAge** is created, it **points to the same address as age**
* When **age is re-assigned to 31**, a new piece of memory is created to store the value of 31

**Reference Types**

* For Reference Types, JavaScript create a **unique identifier** which points to **a memory address in the call stack**, which in turn points to a **memory address in the HEAP**
* Why? Because Object may be **too large to store in the stack** – HEAP is an almost **unlimited memory pool**
* **Example**
* **Friend identifier** will point to the **same memory address** as the **me identifier** in the stack
* That memory address in the stack points to the object HEAP, so, the friend object is **exactly the same** as the me object
* So, any changes to the **me or friend object** will be **reflected in both objects**, as they are **essentially the same object**
* Even though the friend object is assigned as const, it can still be **mutated** because the address in the call stack is not changing – only the value in the **HEAP is changing**
* So, all **primitive values** declared with const are **immutable** and all **reference values** are **mutable**

**More Examples**

**Primitive Types**



Everything works intuitively, because each **primitive value** will simply be saved into **its own address in memory in the stack**

**Reference Types**

Graphical user interface, application, Teams

Description automatically generated

With reference types, the marriedJessica object is created but is pointing to **exactly the same address in the HEAP** as the Jessica object, so when the lastName has been mutated on the Jessica object, the change effects **both objects**

marriedJessica is **not a new object in the heap**, it’s simply another variable in the stack which holds a reference to the object in the heap

**Trying to re-assign const marriedJessica**

Graphical user interface, application, Teams

Description automatically generated

Trying to re-assign the marriedJessica object to an empty object **will not work**, as marriedJessica is a const variable and **const variable cannot have their values changed to a new memory address in the stack**

Therefore, re-assigning the property of an object is completely different to re-assigning it because of how memory storage works with the stack/HEAP

**Creating a shallow copy/clone of an object**

**Graphical user interface, application, Teams

Description automatically generated**

* Object.assign **merges 2 objects** then **returns a new object** – in this case, an **empty object is merged with jessica2**
* jessicaCopy re-assigns lastName property to Davis, and **JessicaCopy has its own memory address in the stack**, the change **doesn’t reflect on jessica2**
* **NOTE:** Object.assign only works **on the first level,** meaning if there are object types inside the object, the **inner objects (objects, arrays, functions etc)** will still be the same (point to the same address in the HEAP stack) **-**>object.assign creates a **shallow copy**
* A **shallow copy** only copies the properties in the first level

**A screenshot of a computer

Description automatically generated with medium confidence**

* Line 28 – re-assigning jessicaCopy.family[0] to another name. The change is reflected in **both objects**… only a shallow clone!