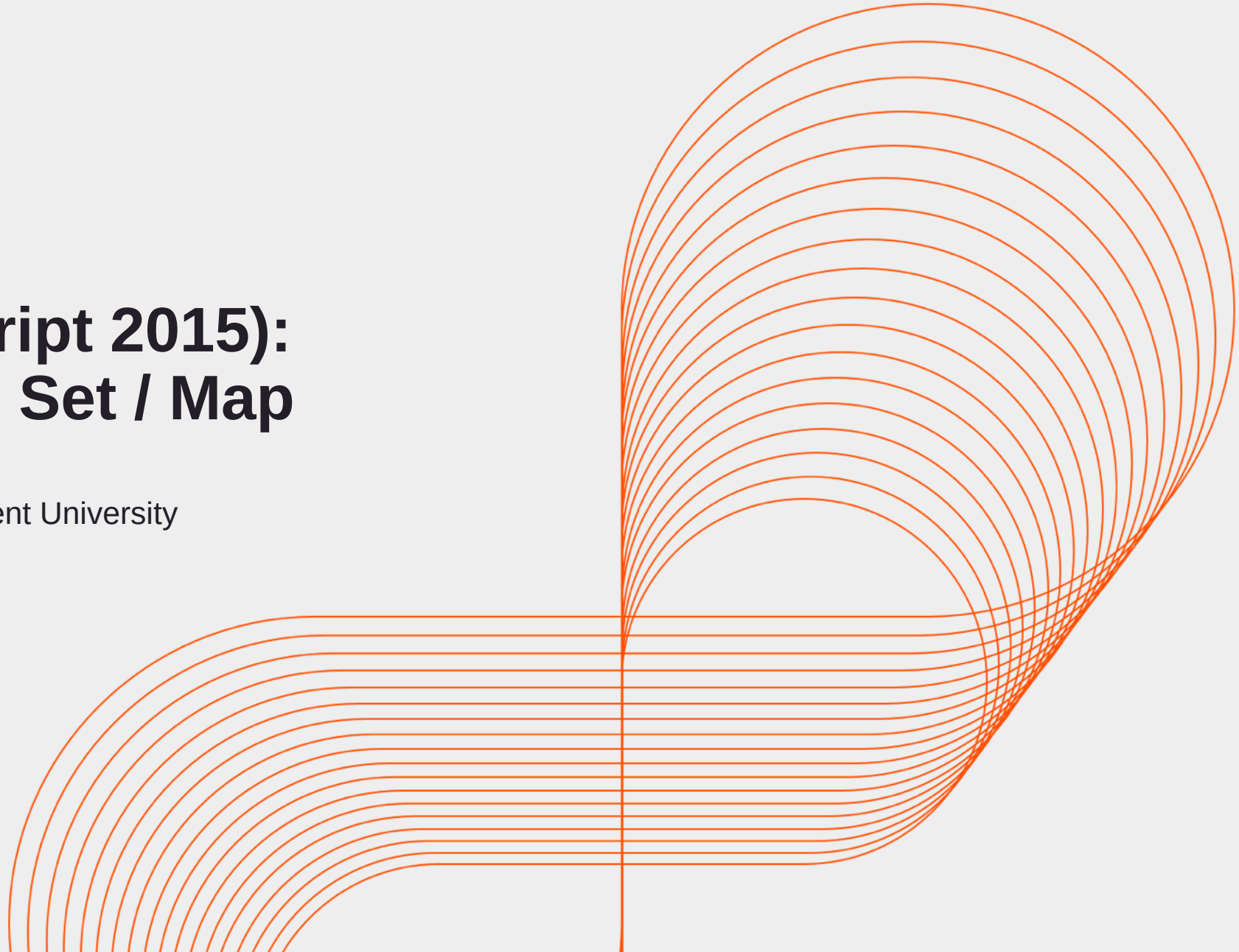
Persistent

ES6 (ECMAScript 2015): Destructuring, Set / Map

Persistent Interactive | Persistent University



Key Learning Points

- Introduction to destructuring.
- Why destructuring is important.
- Array Matching
- Object matching – Shorthand Notation & Deep Matching
- Object & Array Matching, Default Values
- Parameter Context Matching
- Fail-Soft Destructuring
- Set Data-Structure
- Map Data-Structure

What is Destructuring?

- Destructuring is the process of structuring a new object or array from an existing object or array.
- It literally destructs the existing object by extracting only values of interest.
- It is done by pattern matching in ES6. It uses an array as data item from which you extract values and assign to a variable according to the pattern used to match the values you need from the array.

Why is Destructuring Important ?

- In ES5 and earlier versions, it was difficult to fetch information from objects and arrays because doing so creates a lot of duplicate code to retrieve certain data from local variables. E.g.

- This code extracts the values of val1 and val2 from the object obj and stores the extracted data in local variables with the same names

```
let obj= {
        val1: true,
        val2: false
};

let val1= obj.val1        //extract data from the object

val2= obj.val2;
```

Why is Destructuring Important ?...

- If the above scenario has to be repeated for many variables and if you must traverse a nested data structure to find the information, it will then become very complex to handle such data extraction.
- That’s why ES6 introduced this destructuring concept for objects and arrays to break down larger and complex data structure and getting the required information.

Array Matching…

- Array Matching is done to pull out the values out of an array and use them as stand-alone variables.
- In Array Matching, the array pattern should be at the left of the assignment and the array being DE structured on the right.

```
const vehicles=[‘car’ ,’bike’, ‘auto’];

const [ fourWheeler, twoWheeler, threeWheeler ] =
vehicles;

console.log(“twoWheeler: ” + twoWheeler);


    //output

    //twoWheeler: bike
```

Object Matching, Shorthand Notation

- Object Matching is done same as array matching.
- An object pattern is to be created at the left side of assignment and also provide the property name whose value you wish to extract.
- Object pattern must have a variable in exactly the same position as the property value to be extracted.

```
var  { foo , bar } = { foo: 'Hello' , bar: 'ES6' }
console.log(bar);

    //output
    ES6
```

Object Matching, Deep Matching

- To extract some specific properties out an object, Deep Matching is used to make it handy. For Example:

```
let cust= { name:“Rahul",
        address:{
                street:"11,Erandwane" ,
                city:"Pune",
                state:"Maharashtra“  }
        };
let{ address: {city:city}, address:{state:state}}=cust;
console.log("City:",city,"\nState;", state);

        //Output
        //City:Pune
        //State:Maharashtra
```

Object and Array Matching , Default Values

- If the value unpacked from the object is undefined, a variable can be assigned a default value in that case. E.g

```
var  {x = 99, y = 11} = { x : 55 };

console.log(x);                 //55
console.log(y);                 //11
```

Parameter Context Matching

- Destructing of Arrays and Objects into an individual parameters during function calls can be done by Parameter Context Matching in ES6. E.g

```
function foo ([ name, age ]) {
        console.log(name , age)
}
function bar ({name : x, age : y}) {
        console.log(x, y );
}
function zen ({name , age}) {
        console.log(name, age);
}
```

Parameter Context Matching…

- Destructing of Arrays and Objects into an individual parameters during function calls can be done by Parameter Context Matching in ES6. E.g

```
foo({ “Sachin”, 22])
bar({name: “Rahul” , age: 7})
zen({name: “Sachin”, age:22})

        //Output
        //Sachin 22
        //Rahul 7
        //Sachin 22
```

Fail-Soft Destructuring

- Fail-soft Destructuring provides optional default values, while trying to pull the values.

- In example, values if available in the ‘arr’ are provided, else variables are provided with the default value(like 30 provided to ‘c’) or undefined if that default value is also not available.

```
let arr= [ 5, 55];
let [a=10, b=20, c=30, d] = arr ;
console.log(“a:”+a);          //5
console.log(“b:”+b);          //55
console.log(“c:”+c);          //30   - default
console.log(“d:”+d);          //undefined
```

Set Data-Structure

- A set in ES6 is basically an ordered list of values that does not contain duplicates.
- Unlike arrays, you can check for the presence of a value in a set rather than typically accessing an individual item in a set.
- Sets are created using new Set() and you can add values to set by calling add() method.
- size property can be used to check how many items are there in the set.
- Sets do not coerce values to determine the similarity between two elements. So, in a set number 7 and the string “7” are two different items.

Creating Sets

- Given is the example to create a set in ES6.

```
let newSet= new Set();
newSet.add(7);
newSet.add(“7”);
newSet.add(7);      //ignored, because its duplicate
console.log(newSet.size);    //2
```

- You can also add objects in sets. E.g

```
let newSet= new Set(),
key1 = { },
key2 = { };
newSet.add(key1);
newSet.add(key2);
console.log(newSet.size);                    //2
```

Set Data-Structure

- A set in ES6 is basically an ordered list of values that does not contain duplicates.
- Unlike arrays, you can check for the presence of a value in a set rather than typically accessing an individual item in a set.
- Sets are created using new Set() and you can add values to set by calling add() method.
- size property can be used to check how many items are there in the set.
- Sets do not coerce values to determine the similarity between two elements. So, in a set number 7 and the string “7” are two different items.

Removing Items from Sets

- You can remove items from sets by calling delete() method.
- All the items in the set can be removed by calling the clear() method. Set will be empty after calling clear() method.
- Given below is the example to delete item from the set.

```
let newSet= new Set();
newSet.add(7);
newSet.add("7");
console.log(set.has(7));                    //true
newSet.delete(7);
console.log(set.has(7));                    //false
```

Converting a Set into an Array

- A Set can be converted into an array by passing the array to the set constructor.
- Converting an array back to a set utilize the spread operator(…) of ES6.
- Spread operator can also be used to convert a set into an array. Given below is the example to convert a set into an array.

```
let newSet= new Set([10,20,30,30]);
newArr= […newSet];
console.log(newArr);                //10,20,30
```

Weak Sets

- Weak sets stores weak object references.
- They cannot store primitive values.
- A weak reference to an object does not prevent garbage collection if it is the only remaining reference.
- Weak sets can be created using a WeakSet constructor.
- Weak sets have all the methods like sets have e.g. add(),delete() and a has() method.
- It does not have a size property.
- It does not have a forEach() method.

Weak Sets…

- Given is the example to create weak set.

```
let newSet = new WeakSet(),
 obj={ };
newSet.add(obj);
console.log(newSet.has(obj));          //true
newSet.delete(obj);
console.log(newSet.has(obj));          //false
```

Map Data-Structure

- A map in ES6 is an ordered collection of keys that corresponds to specific values.
- Each item in a map has two pieces of data, that are keys and values.
- Value of an item can be retrieved from a map by reading its corresponding key.
- Maps in ES6 acts a s cache for storing data that can be utilized later.

Creating Maps

- Maps in ES6 can be created by calling the set() method.
- Values to Maps can be added passing set, a key and the value associate with the key.
- Values can be retrieved from a map by calling get() method and passing key to the method. For example:

```
let newMap= new Map();
newMap.set(“book”, “ECMAScript06”);
newMap.set(“pubYear”, 2017);
console.log(newMap.get(“book”));                    //
“ECMAScript06” console.log(newMap.get(“pubYear”));
                //2017
```

Map Methods

- Maps and sets have may methods in common. For example:
  - has(key): to check if the given key exists in the map/set.
  - delete(key): to remove specific item from map/set.
  - clear():removes all the items from map/set.
- Size property behaves differently in map. As map contains key value pairs, so a key-value pair is considered as one item in case of a Map.

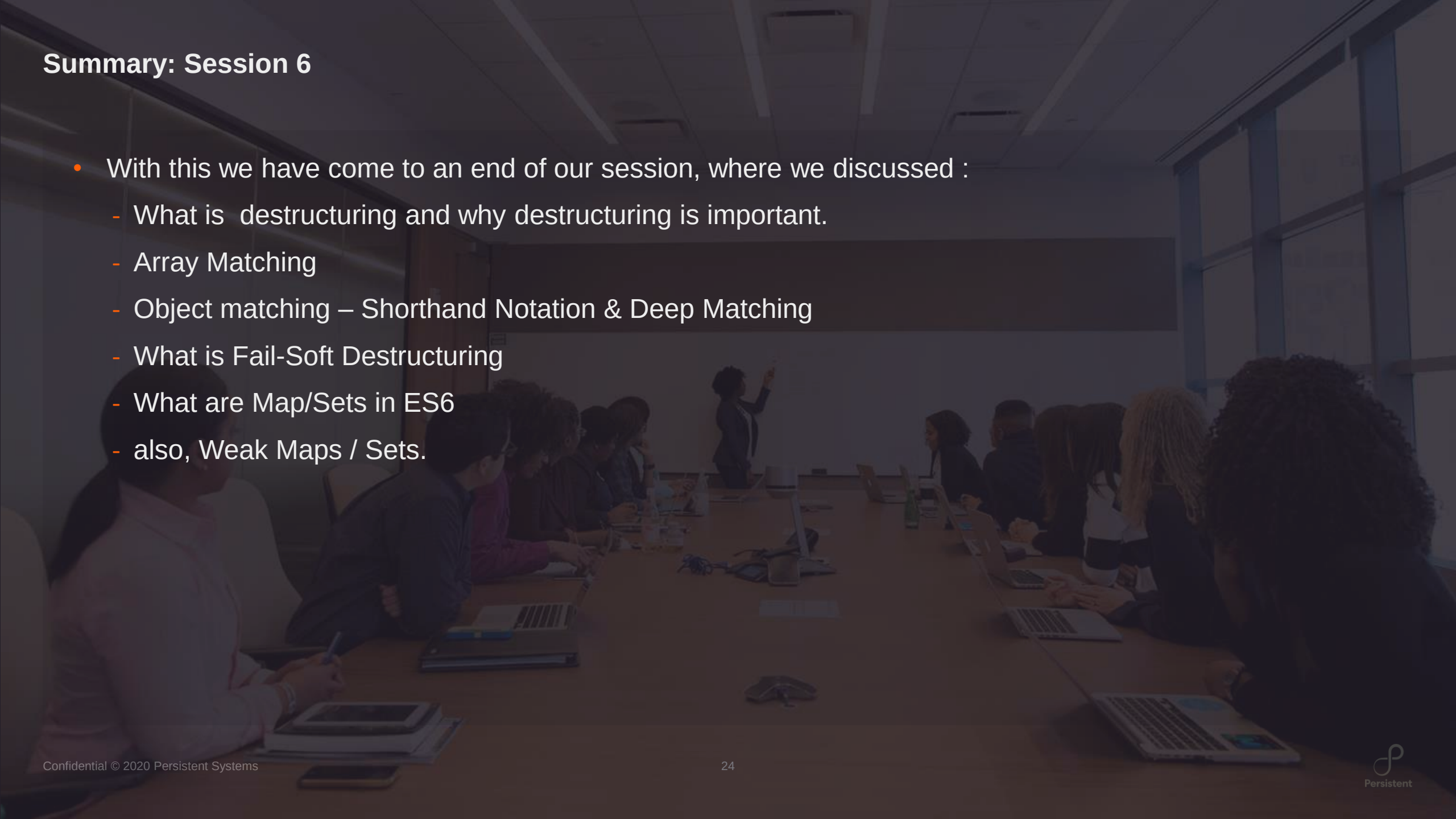
```
let newMap= new Map();
newMap.set(“book”, “ECMAScript06”);
newMap.set(“pubYear”, 2017);
console.log(newMap.size);              //2
console.log(newMap.has(“book”);        //true
newMap.clear();
console.log(newMap.has(“book”);        //false
```

Weak Maps

- Weak maps are used to store weak object references.
- Only weak map keys are weak references not the weak map values.
- Every key in a weak map must be an object. In case a weak map has a non-object key, it will throw an error.
- Weak maps are very similar to Maps and employ the same methods like set() and get() are used to add and retrieve the data
- Like weak sets it does not have size property, so there is no way to check whether the weak map is empty or not.

Summary: Session 6

- With this we have come to an end of our session, where we discussed :
  - What is destructuring and why destructuring is important.
  - Array Matching
  - Object matching – Shorthand Notation & Deep Matching
  - What is Fail-Soft Destructuring
  - What are Map/Sets in ES6
  - also, Weak Maps / Sets.



Appendix

References

Thank You

References

- http://untangled.io/in-depth-es6-destructuring-with-assembled-avengers/
- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- http://es6-features.org
- Understanding ECMAScript 6 by Nicholas C. Zakas
- ECMAScript 6 Succinctly by Matthew Duffield

Persistent

Thank you!

Persistent Interactive | Persistent University