TypeScript Intersection Types

TypeScript **Intersection Types** is in many ways complementary to **Union Types**.

Intersection Types (&)

An intersection type describes a value that has **all the properties of several types combined**. It uses the & (ampersand) symbol. Think of it as merging properties from multiple types into a single new type.

Example:

TypeScript

```
interface Person {
    name: string;
    age: number;
}

interface Employee {
    employeeId: string;
    department: string;
}

// An Admin is a Person AND an Employee
type Admin = Person & Employee;

const admin1: Admin = {
    name: "Alice",
    age: 30,
    employeeId: "E123",
    department: "HR"
};
console.log(admin1.name);         // Valid (from Person)
console.log(admin1.employeeId);   // Valid (from Employee)
// console.log(admin1.salary);    // Error: Property 'salary' does not exist on type 'Admin'.
```

Key Use Cases for Intersection Types:

1. **Composing Types:** You can build complex types by combining smaller, more focused types. This promotes modularity and reusability.
2. **Adding Functionality:** You can add properties or methods to an existing type without directly modifying the original type.
3. **Mixins:** Intersection types are fundamental to implementing mixins in TypeScript, where you combine functionalities from different classes into a single class.

Important Note on Primitive Intersection:

If you intersect two primitive types, the resulting type is never if they are not compatible. For example:

TypeScript

```
type NeverType = string & number; // This results in 'never'
                          // because a value cannot be both a string AND a number simultaneously.
```

However, if you intersect two object types, TypeScript intelligently combines their properties. If there are overlapping properties with different types, the resulting type for that property will be an intersection of those types. For example:

TypeScript

```
interface A {
    x: string;
    y: number;
}

interface B {
    x: number; // 'x' has a different type here
    z: boolean;
}

type C = A & B;
// C would effectively be:
// type C = {
//     x: string & number; // Which simplifies to 'never'
//     y: number;
//     z: boolean;
// }
// So, an object of type C would require an 'x' property that is both a string and a number, which is impossible.
```

Below are few more examples:

Example 1: Creating an intersected type: In this example, two interfaces named student and teacher are created. Intersected type is created by using '&' between student and teacher. Intersected type contains all the properties of the two interfaces. An obj of intersection type is created and values are retrieved from it. We can not use a property without assigning it to the intersection type object.

```
interface Student {
  student_id: number;
  name: string;
}

interface Teacher {
  Teacher_Id: number;
  teacher_name: string;
}

type intersected_type = Student & Teacher;

let obj1: intersected_type = {
  student_id: 3232,
```

```
  name: "rita",
  Teacher_Id: 7873,
  teacher_name: "seema",
};

console.log(obj1.Teacher_Id);
console.log(obj1.name);
```
Output:
7873
Rita

Example 2: In this example, we create two interfaces A and B, in which there are two properties named 'feauA' and 'feauB'. But the type of feauA isn't the same in both the interfaces, when we try to assign a value 20 to feauA typescript compiler raises an error as the intersection type is of the type 'string & number'. If we try to assign a string to feauA, the error is not raised as to when intersected the type is String.

```
interface A {
  feauA: string;
  feauB: string;
}

interface B {
  feauA: number;
  feauB: string;
}

type AB = A & B;

let obj1: AB;
let obj2: AB;

// Error, Type '20' is not assignable
// to type 'string & number'
obj1.feauA = 20;
console.log(obj1.feauA);

obj2.feauB = "c";
console.log(obj2.feauB);
```
Output:
error TS2322: Type 'number' is not assignable to type 'never'.
obj1.feauA = 20;
// Error, Type '20' is not assignable
// to type 'string & number'

Example 3: Intersection types are commutative and associative: The order of the intersection doesn't matter when we intersect two or more types. Even if the order of intersection changes the type of the intersected objects are the same, the 'typeof' operator is used to check that, the properties of the intersected objects are also the same.
commutative property: A&B = B&A
associative property: (A&B)&C = A&(B&C)

```
interface A {
  prop1: String;
}

interface B {
  prop2: String;
}

interface C {
  prop3: String;
}

let obj1: A & B = { prop1: "length", prop2: "width" };
let obj2: B & A = { prop1: "length", prop2: "width" };
let obj3: A & (B & C) = { prop1: "", prop2: "", prop3: "" };
let obj4: (A & B) & C = { prop1: "", prop2: "", prop3: "" };

obj3.prop3 = "height";
console.log(obj3.prop3);

obj4.prop1 = "length";
console.log(obj4.prop1);

console.log(obj3 == obj4); // false
console.log(typeof obj3 == typeof obj4); // true
console.log(typeof obj1 == typeof obj2); // true
```

Output:

```
height
length
false
true
true
```

In summary, both Union and Intersection types are powerful tools in TypeScript's type system, allowing you to express complex relationships between data shapes effectively.