Generic and keyof

This lesson explains how to combine keyof with generic.

You can use `extends` to extend a generic type to another type if you want the generic type to be a subset of the other type.

In the following code, we see that **line 13** expects that the generic `K` extends `T` which means that the second parameter needs to extend the former.

```
interface TypeA {
  prop1: string;
  prop2: number;
}
interface TypeB {
  prop1: string;
  prop3: boolean;
}
interface TypeC extends TypeA {
  prop4: number;
}

function printProps<T, K extends T>(p1: T, p2: K): void { // K must have all field from T at
  console.log(p1);
  console.log(p2);
}

let a: TypeA = { prop1: "p1", prop2: 2 };
let b: TypeB = { prop1: "p1", prop3: true };
let c: TypeC = { prop1: "p1", prop2: 2, prop4: 4 };


// printProps(a, b); // Does not transpile because B does not extends A
printProps(a, c);
```

The commented code at **line 23** does not transpile because `TypeB` is not extending `TypeA`. However, **line 24** is successful because `TypeC` extends `TypeA` at **line 9** which mean that `TypeC` has all its properties merged with `TypeA` properties.

So far, so good but what if we would like to mention that we want to allow the property names of a generic type? For example, let's specify an array of

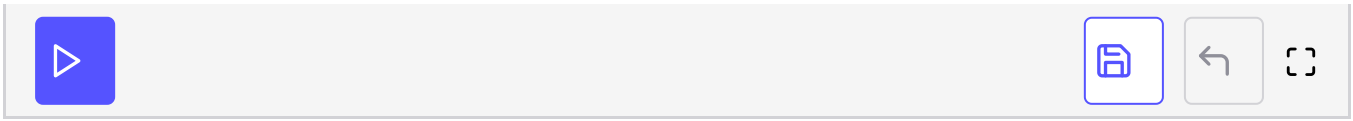
property names to output at the console.

```
interface TypeA {
  prop1: string;
  prop2: number;
}

function printProps<T, K extends keyof T>(p1: T, p2: K[]): void { // Extends all properties'
  console.log("Printing:");
  p2.forEach(propName => {
    console.log(`Name: ${propName} and value: ${p1[propName]}`);
  });
}

let a: TypeA = { prop1: "p1", prop2: 2 };

printProps(a, ["prop1"]);
printProps(a, ["prop1", "prop2"]);
// printProps(a, ["not", "working"]);
```



Line 6 takes any kind of object, regardless of its type under the generic `T`. The second parameter is an array of `K` type. `K` as the constraint of `keyof T`. The `keyof` returns all the property names of T, it is a list of string. By extending `keyof`, TypeScript limits the possible strings to be of the property of the specified type, in that case, `T`. In the example, the only strings that can be passed in the array are `"prop1"` and `"prop2"`. If we remove one or all of the properties of `TypeA`, suddenly, TypeScript will stop transpiling and say that the code is reaching properties that do not exist.

Thus, even if we are specifying strings, we are still strongly typed and protected from mistakes.