Optional properties can be dangerous

What's an optional property in Typescript?

```
type User = {
  email: string; // mandatory property
  cancellationDate?: string; // optional property
};

const chris = {
  email: "chris@example.com",
  cancellationDate: "2024-10-10",
} satisfies User;

const bob = {
  email: "bob@example.com",
} satisfies User;
```

So what's the issue?

Using the **undefined** keyword.

```
const bob = {
  email: "bob@example.com",
  cancellationDate: undefined, // might be a compiler error
} satisfies User;
```

When is it an error?

Depends on the TS config setting exactOptionalPropertyTypes:

- **false**(default value): **allows** assigning `undefined` to optional properties
- **true**: assigning `undefined` to optional properties results in a **compiler error**

Example error message:

```
Type '{ email: string; cancellationDate: undefined; }' does not satisfy the expected type 'User'.
Types of property 'cancellationDate' are incompatible.
Type 'undefined' is not assignable to type 'string'.
```

Just don't assign `undefined` to it ;)

More realistic example

```ts
const buildUser = (email: string, cancellationDate?: string): User => {
  // compiler error when exactOptionalPropertyTypes is true
  return {
    email,
    cancellationDate,
  };
};
```

Error message:

```
Type '{ email: string; cancellationDate: string | undefined; }' is not assignable to type 'User' with 'exactOptionalPropertyTypes: true'.
Consider adding 'undefined' to the types of the target's properties.
  Types of property 'cancellationDate' are incompatible.
    Type 'string | undefined' is not assignable to type 'string'.
      Type 'undefined' is not assignable to type 'string'.
```

Avoiding the error #1

```
const buildUser = (email: string, cancellationDate?: string): User => {
  if (cancellationDate) {
    return {
      email,
      cancellationDate,
    };
  }
  return {
    email,
  };
};
```

- 😀 No compiler error
- 😟 Assigning `email` is duplicated.
- 😟 Adds complexity.

Avoiding the error #2

```
const buildUser = (email: string, cancellationDate?: string): User => {
  const u: User = {
    email,
  };
  if (cancellationDate) {
    u.cancellationDate = cancellationDate;
  }
  return u;
};
```

- 😀 No compiler error
- 😀 Assigning `email` isn't duplicated.
- 😟 Adds complexity.

Avoiding the error #3

The lazy way using the Non-null assertion operator

```
const buildUser = (email: string, cancellationDate?: string): User => {
  return {
    email,
    cancellationDate: cancellationDate!,
  };
};
```

- 😀 No compiler error
- 😀 Assigning `email` isn't duplicated.
- 😀 Doesn't add explicit code complexity.
- 😯 Creates an object which doesn't match the type.

Explaining #3

```
const buildUser = (email: string, cancellationDate?: string): User => {
  return {
    email,
    cancellationDate: cancellationDate!,
  };
};
const bob = buildUser("bob@example.com", undefined); // not a compiler error
console.log(bob);
```

`bob` will be:

```
{
  "email": "bob@example.com",
  "cancellationDate": undefined
}
```

`bob` is literally not an object of type `User`!

Explaining #3

What's the issue with `bob` not being a `User`?

Explaining #3

It start's when other parts of the code base trust the compiler!

Explaining #3

```
const hasCancelled = (u: User): boolean => {
  return "cancellationDate" in u;
};
```

Correct implementation regarding the type declaration. The type declares that `undefined` won't happen.

But imagine the result for bob which was created using the Non-null assertion operator.

```
console.log(hasCancelled(bob)); // true
```

Wrong! Bob hasn't cancelled and will call customer service.

Suggestion #1

The Non-null assertion operator should be used more carefully! It's not a `It's going to be fine` operator. It means: "Hey compiler, you cannot know, but i did ensure in a way you cannot see that it's never going to the undefined".

Suggestion #2

Avoid enabling exactOptionalPropertyTypes in new projects/packages as long as you don't have reason.

Suggestion #3

If you are working on a project with `exactOptionalPropertyTypes: true`, avoid optional properties on types/interfaces without also declaring `undefined`:

```
type User = {
  email: string;
  cancellationDate?: string | undefined; // added undefined
};
const buildUser = (email: string, cancellationDate?: string): User => {
  return {
    email,
    cancellationDate, // no condition, no incorrect non-null usage
  };
};
const hasCancelled = (u: User): boolean => {
  return u.cancellationDate !== undefined;
};

const bob = buildUser("bob@example.com", undefined); // not a compiler error
console.log(bob); // { "email": "bob@example.com", "cancellationDate": undefined }
console.log(hasCancelled(bob)); // false
```

Suggestion #4

When using the `in` operator to assert the presence of an optional property on an object, think twice if you should assert for not `undefined` instead.

Thank you