TypeScript Code Safety and Optimization

Strict Mode

One of TypeScript's most powerful features is its strict type-checking capabilities, which can help catch potential bugs early in the development process. By enabling strict mode, you instruct TypeScript to perform more comprehensive checks, ensuring that your code adheres to safer practices.

To enable strict mode, you need to modify your `tsconfig.json` file and set the `"strict"` option to true. This activates a set of type-checking features that can drastically improve your code quality.

```
// tsconfig.json
{
  "compilerOptions": {
     "strict": true
   }
}
```

Enabling this option might result in more errors, but it helps you write more robust, bug-free code.

No Implicit `any`

By default, TypeScript attempts to infer types wherever it can. However, sometimes it doesn't have enough information to do so and falls back to the any type. This can be dangerous because any disables type safety, meaning TypeScript will not check what you're doing with that variable.

Consider this function:

```
function logName(a) {
  // No error here??
  console.log(a.name);
}

logName(97);
```

Here, the parameter a is implicitly typed as any, which means TypeScript will not complain if we pass in something unexpected like a number. If a is a number, trying to access the `name` property will result in `undefined`, but no error will be raised. This can cause unpredictable behavior.

When "`noImplicitAny`" is enabled through strict mode, TypeScript will require you to explicitly define the type of a:

```
// Error: Parameter 'a' implicitly has an 'any' type.
function logName(a: any) {
  console.log(a.name);
}
```

You can fix this by explicitly stating the type or using a more appropriate one:

```
function logName(a: { name: string }) {
  console.log(a.name);
}

logName({ name: 'John' }); // Works fine
```

This change forces you to think about the shape and type of the data your function will handle, improving type safety.

Strict Null Checks

In JavaScript, `null` and `undefined` are common sources of runtime errors. Without strict null checks, TypeScript treats `null` and `undefined` as acceptable values for any type, leading to possible issues when accessing properties or methods on values that don't exist.

Here's an example:

```
let whoSangThis: string = getSong();

const singles = [
  { song: 'touch of grey', artist: 'grateful dead' },
  { song: 'paint it black', artist: 'rolling stones' },
];

const single = singles.find((s) => s.song === whoSangThis);
```

```
console.log(single.artist); // What if `single` is undefined?
```

In this code, `singles.find` might not find a match, which would return `undefined`. However, we're assuming that it always finds something and directly access the `artist` property, which could lead to a runtime error.

With `"strictNullChecks": true`, TypeScript won't let you make such assumptions. You'll get an error if you try to access properties on a value that could potentially be `null` or `undefined`.

```
// ERROR: Object is possibly 'undefined'.
console.log(single?.artist);
```

To fix this, we need to add a check to ensure that `single` is defined before accessing its properties:

```
if (single) {
  console.log(single.artist); // Now safe
}
```

This forces you to handle cases where `null` or `undefined` might sneak in, making your code more resilient to unexpected failures.

Extra Features in Strict Mode

In addition to the two major checks we've discussed, strict mode enables a few more useful features that improve type safety across your code:

1. Strict Function Types

   Strict function types ensure that functions are checked more rigorously when passed as arguments to other functions. TypeScript will enforce that function parameters and return types align correctly, preventing common mismatches.

   Example:

   ```
   function callWithNumber(callback: (n: number) => void) {
     callback(42);
   }
   ```

```
callWithNumber((x) => console.log(x.toFixed(2))); // Good
callWithNumber((x) => console.log(x.length)); // Error: 'number' has no
                                                  'length' property
```

2. Always Strict Mode

When strict mode is enabled, TypeScript always operates in strict JavaScript mode. This means it will perform additional runtime checks and behave in a more predictable manner in terms of scoping and reserved words.

3. Strict Property Initialization

This feature ensures that all properties of a class are initialized when an instance is created. This can prevent issues where certain properties are accidentally left undefined due to forgotten assignments.

Example:

```
class User {
  name: string;
  age: number; // Error: Property 'age' has no initializer and is not
                  definitely assigned in the constructor

  constructor(name: string) {
    this.name = name;
  }
}
```

This error can be resolved by either initializing age or marking it as optional with ?:

```
class User {
  name: string;
  age?: number;

  constructor(name: string) {
    this.name = name;
  }
}
```

Summary

Strict mode in TypeScript helps enforce stricter type-checking rules, which leads to fewer bugs and more maintainable code. By catching potential issues early, such as unhandled null values or missing property initializations, you can write cleaner, safer, and more reliable code.

To summarize, enabling strict mode enables:

- **No implicit `any`**: Requires explicit typing and prevents unsafe code from creeping in.
- **Strict null checks**: Forces you to handle `null` and `undefined` explicitly, making your code more resilient.
- **Strict function types**: Ensures function arguments and return types are always correctly typed.
- **Strict property initialization**: Enforces class property initialization to prevent `undefined` values.

Strict mode may seem like it generates more errors, but those errors are helping you spot potential issues before they arise in production. It's a highly recommended setting for any TypeScript project.

Narrowing

In TypeScript, it's common to work with variables that can hold multiple types, such as `string | number` or custom union types. **Type narrowing** is the process of refining the type of a variable from a broader type (like a union) to a more specific type based on runtime checks.

Narrowing is essential because it allows TypeScript to understand which specific type a variable holds at a given point in the code, enabling you to perform operations safely and take advantage of TypeScript's powerful type system.

Basic Narrowing with `typeof`

A common way to narrow types in TypeScript is by using the `typeof` operator. This is useful when dealing with primitive types like `string`, `number`, and `boolean`. Here's a simple example:

```
function repeatValue(val: string | number): string {
  if (typeof val === 'string') {
    // Since TypeScript knows `val` is a string here, we can call string methods
    return val.repeat(2);
  }
  // Otherwise, `val` must be a number, so we handle that case
  return (val * 2).toString();
}

console.log(repeatValue('Hello')); // "HelloHello"
console.log(repeatValue(10));      // "20"
```

In this example, TypeScript uses `typeof` to narrow the `val` type from `string | number` to either `string` or `number` within the corresponding `if` and `else` blocks, making it safe to use type-specific methods.

Narrowing with `in` Operator

The `in` operator is useful when you're working with objects that share some properties but differ in others. Here's an example where we have two types, `Car` and `Bicycle`, and we use the `in` operator to narrow the type:

```
interface Car {
  type: 'Car';
  horsepower: number;
  doors: number;
}

interface Bicycle {
  type: 'Bicycle';
  gears: number;
}

type Vehicle = Car | Bicycle;

function describeVehicle(vehicle: Vehicle): string {
  if ('horsepower' in vehicle) {
   // TypeScript now knows `vehicle` is of type `Car`
   return `Car with ${vehicle.horsepower} horsepower and ${vehicle.doors} doors.`;
  }
  // Otherwise, it's a `Bicycle`
  return `Bicycle with ${vehicle.gears} gears.`;
}

const car: Car = { type: 'Car', horsepower: 150, doors: 4 };
const bike: Bicycle = { type: 'Bicycle', gears: 21 };
console.log(describeVehicle(car));  // "Car with 150 horsepower and 4 doors."
console.log(describeVehicle(bike)); // "Bicycle with 21 gears."
```

Here, `in` checks for the presence of a property (`horsepower`) that only exists on the `Car` type, allowing TypeScript to narrow `vehicle` from `Vehicle` (a union type) to `Car`.

Literal Type Narrowing with Discriminated Unions

Discriminated unions use a common literal property to distinguish between types. This is a more structured approach to narrowing, often used in situations where types need to share a key but hold different values:

```
interface Dog {
  kind: 'Dog';
  barkVolume: number;
}

interface Cat {
  kind: 'Cat';
  meowVolume: number;
}

type Pet = Dog | Cat;

function petNoise(pet: Pet): string {
  if (pet.kind === 'Dog') {
    // TypeScript narrows `pet` to `Dog` type here
    return `The dog barks at volume: ${pet.barkVolume}`;
  }

  // TypeScript automatically narrows `pet` to `Cat` type here
  return `The cat meows at volume: ${pet.meowVolume}`;
}

const dog: Dog = { kind: 'Dog', barkVolume: 80 };
const cat: Cat = { kind: 'Cat', meowVolume: 50 };

console.log(petNoise(dog)); // "The dog barks at volume: 80"
console.log(petNoise(cat)); // "The cat meows at volume: 50"
```

The property `kind` is a literal value (`'Dog'` or `'Cat'`), which helps TypeScript to differentiate between the types and allows the compiler to infer the correct type within the `if` block.

Narrowing with `instanceof`

When working with classes or custom objects, you can use the `instanceof` operator to narrow down the type of an object. This works when you need to check if an object is an instance of a specific class:

```
class Fish {
  swim() {
    console.log('Fish is swimming');
  }
}

class Bird {
  fly() {
    console.log('Bird is flying');
  }
}

type Animal = Fish | Bird;

function moveAnimal(animal: Animal) {
  if (animal instanceof Fish) {
    // TypeScript knows `animal` is a Fish
    animal.swim();
  } else {
    // TypeScript knows `animal` is a Bird
    animal.fly();
  }
}

const goldfish = new Fish();
const parrot = new Bird();

moveAnimal(goldfish); // Fish is swimming
moveAnimal(parrot);   // Bird is flying
```

In this example, `instanceof` helps TypeScript recognize whether the `animal` is a `Fish` or `Bird`, allowing the appropriate method (`swim` or `fly`) to be safely invoked.

Custom Type Guards

For more complex type narrowing scenarios, you can create your own **custom type guards**. These are functions that help TypeScript understand what type a variable holds by returning a boolean value and using the `is` keyword to specify the type:

```
interface Dog {
  bark: () => void;
}

interface Cat {
  meow: () => void;
}

function isDog(pet: Dog | Cat): pet is Dog {
  return (pet as Dog).bark !== undefined;
}

function petSound(pet: Dog | Cat) {
  if (isDog(pet)) {
    pet.bark(); // TypeScript now knows pet is a Dog
  } else {
    pet.meow(); // TypeScript now knows pet is a Cat
  }
}

const dog: Dog = { bark: () => console.log('Woof!') };
const cat: Cat = { meow: () => console.log('Meow!') };

petSound(dog); // "Woof!"
petSound(cat); // "Meow!"
```

In this example, the custom type guard `isDog` helps TypeScript determine whether the `pet` is a `Dog`. Inside the `if` block, TypeScript narrows the type and allows us to use `Dog`-specific methods.

Exhaustiveness Checking with `never`

When working with union types, it's a good idea to ensure all possible cases are handled. You can use the `never` type in combination with `switch` statements to enforce exhaustiveness:

```
interface Square {
  kind: 'square';
  size: number;
}

interface Circle {
  kind: 'circle';
  radius: number;
}
```

```
type Shape = Square | Circle;

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case 'square':
      return shape.size * shape.size;
    case 'circle':
      return Math.PI * shape.radius * shape.radius;
    default:
      const _exhaustiveCheck: never = shape;
      throw new Error(`Unknown shape: ${_exhaustiveCheck}`);
  }
}

const square: Square = { kind: 'square', size: 10 };
const circle: Circle = { kind: 'circle', radius: 5 };

console.log(getArea(square)); // 100
console.log(getArea(circle)); // 78.5398...
```

The `never` type in the default case helps ensure that all possible cases in the `Shape` union are handled. If a new shape is added but not handled in the switch, TypeScript will raise an error during compilation, helping you avoid runtime issues.