

Principles of Programming Languages

Type Checking

Adding types to a program has three key advantages:

- It allows the compiler to detect errors that would otherwise only be detected at runtime. It is much better to detect errors as early as possible in the development cycle.

- It serves as excellent documentation by reflecting the intention of the programmer.

- More importantly, it helps the programmer **model** the solution she is designing. A good typing system drives programs towards systematic models - both for data and for code.

Adding types is sometimes “annoying” because it can make programs longer, and for complicated types or generic types - it can make simple programs complex. For example, a function such as $x \Rightarrow x$ (known as the identity function) has a complex type (because it can work on any possible types). We will see further examples of this later.

TypeScript alleviates these 2 problems by:

- Making type annotations optional.
- Making type annotations implicit (that is, where the type of expressions can be inferred from the code, it will be).

This approach is called **gradual typing**. As a general strategy in programming languages, gradual typing allows programmers to trade-off prototyping and type safety.

Consider the following type definition and the function operating over values in this type:

Runtime Errors Caused by Unexpected Values

```
interface Date {  
    year: number;  
    month: number;  
    day: number;  
}
```

```
interface Person {  
    birthDate: Date;  
    name: string;  
}
```

Runtime Errors Caused by Unexpected Values

```
import { map } from "ramda";  
  
const computeAges =  
  people => map(p => 2021 - p.birthDate.year,  
                people);
```

Runtime Errors Caused by Unexpected Values

```
let people = [{ name: "Ben",  
                 birthDate: { year: 1989,  
                             month: 1,  
                             day: 8 } },  
              { name: "Itay",  
                 birthDate: { year: 1994,  
                             month: 8,  
                             day: 17 } }];  
computeAges(people); // => [ 32, 27 ]
```

Runtime Errors Caused by Unexpected Values

Consider now what happens if we invoke `computeAges` on this value:

```
let people = [{ name: "Ben" },  
               { name: "Itay" }];  
computeAges(people);  
// => TypeError: Cannot read property 'year'  
//               of undefined
```

Runtime Errors Caused by Unexpected Values

We obtain a runtime error when we try to access a property of a map which is not provided in the actual value passed to the function. In other words, there is an incompatibility between the value passed to the function and the type of the value expected by the function.

Type annotations are introduced in programming languages to enable **type checking**. Type checking is a process that analyzes a program and verifies that such incompatibilities between variables or function parameters and the values to which they are bound at runtime are impossible for all possible executions of the program.

When type checking is performed at compilation, it detects errors that could otherwise occur later at runtime. This is a much better situation - errors caught early are errors that are avoided. Errors caught at runtime cause damage.

Compilers that implement type checking provide an extremely powerful form of program verification.

Why Would Anyone Give Up on Type Checking?

Given that type checking is such a powerful tool
- why would any programmer want to give up
this and use a programming language that does
not provide it?

Why Would Anyone Give Up on Type Checking?

The arguments usually advanced by proponents of languages without type checking are the following:

Why Would Anyone Give Up on Type Checking?

- **Conciseness:** Programs without type annotations are more concise. Conciseness is important because it leads to more readability and better understanding. For example, the function above with types looks like:

```
const computeAges: (ps: Person[]) => number[] =  
  ps => map(p => 2021 - p.birthDate.year,  
            ps);
```

Why Would Anyone Give Up on Type Checking?

This argument is quite weak - because the additional type annotation is in fact a form of documentation which clarifies the intention of the programmer.

For example, consider the untyped function:

```
const add = (x, y) => x + y;
```

Why Would Anyone Give Up on Type Checking?

In JavaScript, the `+` operator works on a wide range of values (number, strings, arrays). It often produces unexpected output for example:

```
[1, 2] + ["a", "b"]; // => 1,2a,b
```

Why Would Anyone Give Up on Type Checking?

If a programmer declares that the intention of this function is to operate only over number, it clarifies the way the function can be tested and what it means for the function to be correct:

```
const add: (x: number, y: number) => number =  
  (x, y) => x + y;
```

Why Would Anyone Give Up on Type Checking?

- Untyped languages are easier to adjust to incremental changes.

Why Would Anyone Give Up on Type Checking?

For example, consider a program that operates over values of type **Person** as described above.

If later, after the program has been put in production, we start introducing additional data for some persons - so that we add a new property **address**.

Why Would Anyone Give Up on Type Checking?

A program in an untyped language can be “patched” incrementally to support such values - by adding runtime checks for the presence of the additional property. Such patching would require much more in-depth modifications in a statically type-checked language.

Why Would Anyone Give Up on Type Checking?

- Untyped languages encourage interactive programming.

Why Would Anyone Give Up on Type Checking?

When programmers interact with an interpreter, interactively, try a version of a function, then revise it, run tests, modify the structure of the values, revise the functions in fast experimentation cycles, with no compilation and direct interpretation - untyped languages allow faster experimentation.

Why Would Anyone Give Up on Type Checking?

The counter argument is once the programmer has completed the experimentation / prototyping phase, the second stage of *cleaning up* with type definition can only improve the quality of the program and its design. Gradual typing is attractive because it supports both types of operations and transition from prototype to typed correct versions.

Why Would Anyone Give Up on Type Checking?

There are other arguments in favor of untyped languages which we will not develop further that are related to the type of polymorphism that is encouraged by each style. In this course, we **strongly adopt the typed style of programming** (we have chosen our camp 😊).

TypeScript adds optional type declarations to JavaScript.

- Type declarations are optional. If they are present, they are checked, otherwise no check is performed. This means that regular JavaScript with no type annotations at all are valid TypeScript expressions.

- The TypeScript compiler performs two tasks:
 - (1) It translates a TypeScript program into a JavaScript program
 - (2) It checks that the program satisfies all the type declarations specified in the program.

- Type annotations can be implicit and inferred by the TypeScript compiler.

The following TypeScript expression:

```
const add: (a: number, b: number) => number =  
  (a, b) => a + b;  
add(1, 2);
```

is translated by **tsc** (the TypeScript compiler)
into:

```
const add = (a, b) => a + b;  
add(1, 2);
```

Type Annotations

Type annotations are optional in TypeScript.

They can occur in the following contexts:

- After variable declarations:

```
let x: <type> = ...
```

- As part of a function signature:

```
(param: <type>, ...) => ...
```

```
function f(param: <type>, ...): <type> {  
    ...  
}
```

Type annotations are written in a specific form - which is called the **type language**. The type language is used to describe the expected type of variables, parameters or the value returned by functions.

In general, all type expressions in the type language are used to refer to a specific set of values. The relation between a type expression and a set of values to which it corresponds is called **denotation** (The denotation relation is in general the relation that exists between a *name* and the *object* to which the name refers).

The simplest type language expression refers to a primitive type - for example **number**, **boolean**, **string**.

It is used as follows:

```
let x: number = 2;  
let a: string = "a";  
let b: boolean = true;
```

These atomic type expressions denote the primitive sets of values that are defined in JavaScript (set of all number values, set of all string values, set of all boolean values).

More complex type language expressions are needed to describe types over compound values. These compound type expressions are constructed recursively by combining smaller type expressions.

To describe a homogeneous array, the following notation is used: `<type>[]`.

```
let stringArr: string[] = ["a", "b", "c"];
```

Array Type Expressions

The following values are not compatible with this type annotation:

```
stringArr = ["a", 1];  
// => Type 'number' is not assignable  
//      to type 'string'.  
stringArr[0] = true;  
// => Type 'true' is not assignable  
//      to type 'string'.
```

Map Type Expressions

To describe map types, the following notation is used: { <key>: <type>; ... }.

```
let p: { name: string, cs: boolean, age: number } =  
  { name: "Ben", cs: true, age: 32 };
```

The expected meaning of the map type description is that **all the keys in the annotation must be present in the value with the described type.**

There could be more keys (as illustrated in the next example) - but all the keys that are specified in the type must be present.

Map Type Expressions

```
let p = {  
  name: "Ben",  
  cs: true,  
  gender: "male",  
  age: 32  
};
```

```
let p2: {name: string, cs: boolean, age: number} =  
  p;
```

Map Type Expressions

Note, however, that when we bind a variable to a literal expression, the TypeScript compiler demands that the type of the literal be **exactly** that of the variable - with no extension.

```
let p: {name: string, cs: boolean, age: number} = {  
  name: "Ben",  
  cs: true,  
  gender: "male",  
  age: 32  
};  
// => Object literal may only specify  
//     known properties...
```

Map Types Relationships

The map type `{}` denotes all possible map values.

The map type `{ a: string }` denotes all maps that have a key `a` with a **string** value - for example:

`{ a: "x" }, { a: "y", b: 1 }`

and the following values are **not** part of this type:

`{}, { b: 1 }, { a: 1 }`

A map type `{ a: string, b: number }` denotes all maps that have a key `a` with a `string` value and a key `b` with a `number` value.

We infer from this definition that the type `{ a: string, b: number }` is a strict sub-type of the type `{ a: string }`.

What is the relation between the types?

```
type mapAB = { a: string, b: number };  
type mapAC = { a: string, c: boolean };
```

Map Types Relationships

These two types are distinct:

`{ a: "x", b: 2 }` is in `mapAB` and not in `mapAC`;

`{ a: "x", c: true }` is in `mapAC` and not in `mapAB`.

Yet - they have a non-empty overlap:

`{ a: "x", b: 2, c: true }` is in both `mapAB` and `mapAC`.

Map Types Relationships

In general - the intersection of the types `mapAB` and `mapAC` is the type:

```
type mapABC = { a: string, b: number, c: boolean };
```

The following types are disjoint:

```
type mapAS = { a: string };
```

```
type mapAN = { a: number };
```

because the constraints on the value of the key
a are incompatible.

A good way to think about these relations is the following:

- Each **key: type** component of the map type specification is a constraint on the values that belong to the type.

- The more constraints are specified in a type, the less values belong to the type.

- If you join all the constraints in two map type specifications - you obtain a type specification which denotes the intersection of the types (as long as the constraints are compatible).

Type expressions can be given names. For example, a map type expression can be named using the `interface` construct:

```
interface <typeName> {  
    <key>: <type>;  
    ...  
}
```

Other type constructs can be given a name using the **type** construct of the type language:

```
type <typeName> = <type>;
```

Named Type Expressions

```
interface Student {  
  name: string;  
  cs: boolean;  
  age: number;  
}
```

```
let s: Student = {  
  name: "Ben",  
  cs: true,  
  age: 32  
};
```

A type expression can have arbitrary type expressions embedded recursively.

For example:

Embedded Type Expressions

```
interface Course {  
    courseName: string;  
    dept: string;  
    semester: number;  
    year: number;  
};
```

```
interface RegisteredStudent {  
    name: string;  
    cs: boolean;  
    age: number;  
    courses: Course[];  
};
```

Embedded Type Expressions

```
let rs: RegisteredStudent = {  
  name: "Ben",  
  cs: true,  
  age: 32,  
  courses: [{  
    courseName: "PPL",  
    dept: "CS",  
    semester: 2,  
    year: 2021  
  }]  
};
```

Implicit Type Annotations

TypeScript can infer the type of variables based on their usage. This is called an **implicit type annotation**. For example:

```
let x = 5;  
x = "a"; // => Type '"a"' is not assignable  
         // to type 'number'.
```

One can use the following notation to declare that a variable should have the type of another variable:

```
let p1 = { name: "Ben", age: 32 };  
let p2: typeof p1 = { name: "nok" };  
// => Property 'age' is missing in type  
//      '{ name: string; }' but required in  
//      type '{ name: string; age: number; }'.
```

The **typeof** x construct is called a **type query** in TypeScript.

NOTE: The **typeof** x type query annotation is different from the **typeof** operator we discussed in the previous lecture - which is a primitive runtime operator in JavaScript.

The TypeScript **typeof** is a type annotation - it belongs to the Type Language - its value is a type annotation.

The JavaScript **typeof** is a runtime primitive operator. Its value is a string.

Consider the case of defining a binary tree.
Using JSON, we would encode a binary tree as follows:

```
let binTree = {  
  root: 1,  
  left: { root: 2 },  
  right: { root: 3 }  
};
```

The TypeScript type that corresponds to this value would be:

```
{  
  root: number;  
  left: ?;  
  right: ?;  
}
```

Recursive Types

We would need to specify that the **left** and **right** keys expect to receive values of exactly the same type. The only way to achieve such a declaration is to use a **recursive type declaration** – which requires us to give a name to the type:

```
interface BinTree {  
  root: number;  
  left: BinTree;  
  right: BinTree;  
}
```

Let us try to create a value according to this type specification:

```
interface BinTree {  
  root: number;  
  left: BinTree;  
  right: BinTree;  
}
```

```
let tree: BinTree = {  
  root: 1,  
  left: { root: 2 },  
  right: { root: 3 }  
}
```

Why didn't that work?

There are multiple solutions to this problem. One is to indicate that the properties **left** and **right** are optional. This is indicated using the `?` syntax as follows:

```
interface BinTree {  
  root: number;  
  left?: BinTree;  
  right?: BinTree;  
}
```

How different is the type annotation:
`{ a?: string, b: number }` from:
`{ a: any, b: number }?`

If we want to define a binary tree whose nodes can have any type - but where all the nodes in the tree must have the same type, we must introduce **type variables**.

```
interface BinTree<T> {  
    root: T;  
    left?: BinTree<T>;  
    right?: BinTree<T>;  
}
```

```
let tree: BinTree<number> = {  
  root: 1,  
  left: { root: 2 },  
  right: { root: 3 }  
};
```

If we want to use a heterogeneous tree (a tree that can contain nodes of different types), we can use the special type called **any**:

```
let tree: BinTree<any> = {  
  root: 1,  
  left: {  
    root: true,  
    left: { root: "hello" }  
  }  
};
```

The **any** type is compatible with all values - it denotes the set of all possible values. Every type that can be defined is actually a subset of the **any** type. In itself, **any** is not useful - typing a variable as **any** does not indicate any constraint on the variable. But when used as a type parameter in a complex type definition as above, **any** becomes useful.

When are Generic Types Useful?

Generic types are compound type expressions with type variables. Some of their components are thus left as unspecified types. When are such partially unspecified types useful?

When are Generic Types Useful?

- One reason to define a generic type is to enable writing generic functions that operate over all possible instantiations of the generic type in the same manner.

When are Generic Types Useful?

```
const treeNodesNumber: <T>(t: BinTree<T>) => number =  
  t => t === undefined  
    ? 0  
    : 1 + treeNodesNumber(t.left)  
        + treeNodesNumber(t.right);
```

When are Generic Types Useful?

```
const treeDepth: <T>(t: BinTree<T>) => number =  
  t => t === undefined  
    ? 0  
    : 1 + Math.max(treeDepth(t.left),  
                    treeDepth(t.right));
```

When are Generic Types Useful?

- Another reason to use generic types is when we need to define a set of value that is a combination between two types that are dependent of each other. For example, consider the set of values which represent an array of values together with the minimum of the values that appear in the array. Such data structure can be defined for any type T which denotes a set of values over which an order relation exists.

When are Generic Types Useful?

```
interface ArrayMin<T> {  
    min: T;  
    values: T[];  
}
```

What is the type relationship (inclusion, disjointness) between `BinTree<T>` and `BinTree<number>`?

Generic Types Relationships

We first need to realize that the type expression `BinTree<T>` cannot be used as the type of a variable if it is not in the scope of a type variable. That is, the expression:

```
let tree: BinTree<T> = ...
```

is not possible - because `BinTree<T>` cannot be in itself the type of a variable.

A generic type can only appear in the context of a generic function or in the context of a larger generic type.

When the **tree** variable is instantiated to a concrete (non-generic) type, then the type can be used as any other type:

```
let tree: BinTree<string> = {  
    root: "Ben",  
    left: { root: "Itay" }  
};
```

The relation between a generic type such as `BinTree<T>` and a type such as `BinTree<string>` is called **instantiation**.

Now, what is the relation between
`BinTree<string>` and `BinTree<number>`?

These two concrete types are instances of the same generic type. Any value in `BinTree<string>` has a property `root` of type `string`. Any value in `BinTree<number>` has a property `root` of type `number`. We infer that these two types are **disjoint**.

Note that while these two types are disjoint, we still can write generic functions that operate on values of either type using the same code, as we have seen in `treeNodesNumber` and `treeDepth`.

If a type **S** is a subtype of a type **T**, then what is the relation between the types **BinTree<S>** and **BinTree<T>**?

Consider, for example,

$T = \{ \text{name: string} \}$ and

$S = \{ \text{name: string, age: number} \}.$

To answer this question, we can read the definition of **BinTree<T>** as a set of constraints a value must meet to belong to the type this type expression denotes:

- The value must be a map value.
- A property **root** must be present and have a value of type **T** - which means that it must have a property **name** of type **string**.
- Properties **left/right** can either be absent or present and of type **BinTree<T>**.

Knowing that any value of type **S** is also a value of type **T** (by definition of subtyping), we can infer that any value in **BinTree<S>** meets all the constraints to also belong to **BinTree<T>**.

In general, if S is a subtype of T , then `BinTree<S>` is a subtype of `BinTree<T>`.

One can type functions in TypeScript. Let us introduce function types step by step:

An untyped function in JavaScript has the following form:

```
function add(x, y) {  
  return x + y;  
}
```

```
const myAdd = function(x, y) {  
  return x + y;  
};
```

```
const myFatAdd = (x, y) => x + y;
```

Function Types

We can first specify the types of the parameters and the return type, in a way similar to the way it would be done in Java. This applies both to named functions and to anonymous functions.

```
function add(x: number, y: number): number {  
  return x + y;  
}  
  
const myAdd = function(x: number, y: number): number {  
  return x + y;  
};  
  
const myFatAdd = (x: number, y: number): number =>  
  x + y;
```

Let us now write the full type of the function out of the function value:

```
const myAdd: (x: number, y: number) => number =  
  (x, y) => x + y;
```

The type expression:

$(x: \text{number}, y: \text{number}) \Rightarrow \text{number}$

is a **function type**. The values that this type denotes are functions that map a pair of numbers to a number - in other words, functions whose domain is within $\text{Number} \times \text{Number}$ and whose range is within Number .

This notation is called the **function signature** - it combines the information on the type of the parameters, their name and the type of the returned value.

Parameter names are just to help with readability. We could have instead written:

```
const myAdd: (base: number, inc: number) => number =  
  (x: number, y: number): number => x + y;
```

As long as the parameter types align, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part of the function type is the return type. We make it clear which is the return type by using a fat arrow (\Rightarrow) between the parameters and the return type. This is a required part of the function type, so if the function doesn't return a value (which means this is a function that just has a side-effect - no return value), we use the special type **void** instead of leaving it off.

Consider the following function definition:

```
let z = 10;  
const add: (x: number, y: number) => number =  
  (x, y) => x + y + z;
```

The definition of **add** refers to the variable **z**, which is defined outside the body of the function. When this happens, we say that the function **captures** the variable **z** - and the value of the function **add** is called a **closure**. This is because the function *closes the captured variables* together with the function definition.

Consider the following example that indicates why this capture mechanism is part of the definition of the closure:

Closures and Their Type

```
const adder = inc => x => x + inc;  
const add5 = adder(5);  
const add2 = adder(2);  
console.log(add5(10));    // => 15  
console.log(adder(3)(4)); // => 7
```

Let us analyze the definition of the function **adder** - and refactor it step by step using types.

`adder` is a function that accepts one parameter `inc`. What should be the type of `inc`?

We look at where `inc` is used in the body of the function `adder` and find the expression `x + inc`. This expression applies to numbers - we conclude that `inc` should be a `number`.

The signature of the function `adder` should therefore look like:

`(inc: number) => ?`

What is the type of the value returned by
`adder`?

The returned value is `x => x + inc`. This expression is a function - which receives `x` as a parameter and returns `x + inc`.

The type of this returned function is thus:

`(x: number) => number`

Putting parameter and return value together, we obtain the type of the function **adder** as this expression:

```
(inc: number) => (x: number) => number
```

Now consider the capturing of the parameter `inc` when the function `adder` is invoked - in the expression:

```
const add5 = adder(5);
```

The type of `add5` is

```
(x: number) => number
```

`adder(5)` returns a function of one parameter.
When it is invoked, this function adds 5 to its
parameter. How does it know to add 5
specifically?

This is because when the return function is computed, it **captures** the current value of the parameter `inc`. This happens in the computation of this expression:

Compute `adder(5)`:

1. Bind parameter `inc` to 5
2. Compute the value `x => x + inc` \Leftarrow This is when variable `inc` is captured

The key point is that when the value of the function is computed, `inc` is bound to the value

5. Later, when the value `add5` is used, `inc` is not bound anymore to any value - because the scope of its definition has been exited.

Closures and Their Type

If we look back at the type of the closures returned when we compute `adder(5)` - we obtain:

`(x: number) => number`

Note that the closure does not indicate that it depends on another variable (`inc`) - because this is not part of the signature of the closure - it is an **internal** aspect of the closure.

As part of the type checking performed by the TypeScript compiler, one must determine whether **two type expressions are compatible**. This compatibility checking occurs in the following context for example:

- Assume we know the type of a variable `a` to be of type expression `<type_a>`.
- We now compile the following expression:

```
let b: <type_b> = a;
```

At this point, the TypeScript compiler must determine whether `<type_a>` and `<type_b>` are compatible.

- Similarly, assume we know the type of a function f to be

$(x: \text{<type_x>}) \Rightarrow \text{<type_y>}$.

- We now compile the following expression:

let $z: \text{<type_z>} = v; f(z);$.

At this point, the TypeScript compiler must determine whether <type_x> and <type_z> are compatible.

Type compatibility is not symmetric - **T1** is compatible with **T2** means we can substitute a value of type **T2** with a value of type **T1** and still obtain valid expressions. The reverse may not be true.

In general, think of type expressions as expressing **constraints** on the values that belong to the type. When a type expression **T1** expresses **more constraints** than **T2**, then it means the type expression **T1** denotes **less values** than **T2**. It also means all the values in **T1** meet the constraints specified by **T2**.

In short: **T1 is a subtype of T2** if all the values in **T1** satisfy the constraints of **T2**.

This means we can do this:

```
let t1: T1 = ...  
let t2: T2 = t1;
```

Compatibility Rules:

- Primitive type expressions are compatible when they are the same (**number** is compatible with **number**, but not with **string**).
- Primitive type expressions are not compatible with any compound type expressions.

Compatibility Rules:

- Arrays are only compatible with arrays, maps with maps, functions with functions.
- Two array expressions **T1[]** and **T2[]** are compatible when **T1** and **T2** are compatible.

The basic rule of TypeScript for map compatibility is:

x is compatible with y if y has at least the same members as x.

For example:

```
interface Named {  
  name: string;  
}
```

```
let x: Named = { name: "Ben" };  
let y = { name: "Itay", location: "London" };  
x = y;
```

This method of checking type compatibility is called **structural typing**. Structural typing is a way of relating types based solely on their members. This is in contrast with **nominal typing** which we know from Java and C++.

Consider the following code:

```
interface Named {  
    name: string;  
}
```

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

Under structural typing, **Person** is a subtype of **Named** - even though this is not declared by the programmer; this subtyping relation is inferred by the compiler.

In **nominally-typed languages** like Java, the equivalent code would be an error because the **Person** type does not explicitly describe itself as being a subtype or an implementor of the **Named** interface - the programmer does not define explicitly relations between the types.

Function Types Compatibility: Comparing the Types of Functions

While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible is a bit more involved. Let's start with a basic example of two functions that differ only in their parameter lists:

```
let f = (a: number, t: string) => 0;  
let g = (b: number, s: string) => 0;  
g = f; // OK
```

Function Types Compatibility: Comparing the Types of Functions

To check if f is assignable to g , we first look at the parameter list. Each parameter in f must have a corresponding parameter in g with a compatible type. Note that the names of the parameters are not considered, only their types. In this case, every parameter of f has a corresponding compatible parameter with identical type in g , so the assignment is allowed.

Function Types Compatibility: Comparing the Types of Functions

We ignore here the complexity introduced by optional arguments and differing number of arguments - we will simplify by stating that two function parameter lists are compatible if they have the same length and the types. The comparison of the type of each parameter, however, is surprising.

Function Types Compatibility: Comparing the Types of Functions

Let's look at how return types are treated, using two functions that differ only by their return type:

```
let f = () => ({ name: "Ben" });  
let g = () => ({ name: "Ben", cs: true });  
f = g; // OK  
g = f; // Type '() => { name: string; }' is  
      // not assignable to type  
      // '() => { name: string; cs: boolean; }'.
```

The type system enforces that:
the source function's return type be a subtype
of the target type's return type.