Complex Types

```
function classify(n: number) {
  if (n < 0) return "negative";
  if (n > 0) return "positive";
  return "zero";
}
```

Complex Types

```
function classify(n: number): "negative" | "positive" | "zero" {
  if (n < 0) return "negative";
  if (n > 0) return "positive";
  return "zero";
}
```

Literal Types

```
let aFoo: 'foo';
let aTrue: true;
let a42: 42;
let manyFoos: 'foo'[];
```

Literal Types

```
let aFoo: 'foo';
let aTrue: true;
let a42: 42;
let manyFoos: 'foo'[];

// All of these are invalid, uninitialized accesses!
aFoo; aTrue; a42; manyFoos;

aFoo = 'foo'; // Great!
aTrue = false; // Error, false is not assignable to true
manyFoos = ['foo','foo','foo', 'bar'] // 'bar' not assignable to 'foo'
```

Unions

```
function classify(n: number): "negative" | "positive" | "zero" {...}
```

Unions

```
type NumberClass = "negative" | "positive" | "zero";
function classify(n: number): NumberClass {...}
```

Type Narrowing

```
type NumberClass = "negative" | "positive" | "zero";

function isBelowZero(value:  NumberClass) {
  if (value === "positive"){
    // type of value inside this statement is "positive"
  } else {
    // type of value inside this statement is "zero" | "negative"
  }
}
```

Exhaustiveness Checking

```
function isBelowZero(value:  NumberClass): boolean {
    switch (value) {
        case "positive":
            return false;
        case "negative":
            return true;
    }
}
```

Exhaustiveness Checking

```
function isBelowZero(value:  NumberClass): boolean {
    switch (value) {
        case "positive":
            return false;
        case "negative":
            return true;
    }
}

// Error: Function lacks ending return statement and
// return type does not include 'undefined'.
```

Intersections

```
type Named = { name: string };
type Aged = { age: number };

type Human = Named & Aged;

const drew: Human = {
    name: "Drew",
    age: 35
}

drew.name // OK
drew.age // OK
```

Unions of Objects

```
type Tea = {
  style: "green" | "black" | "herbal";
  name: string;
};
type Coffee = {
  roast: "dark" | "medium" | "light";
  name: string;
};

type HotDrink = Tea | Coffee;

function drink(drink: HotDrink) {
  drink.name;
  drink.roast; // Type error!
}
```

Discriminated Unions

```
type Tea = {
  type: "tea"; // Discriminant field
  style: "green" | "black" | "herbal";
  name: string;
};
type Coffee = {
  type: "coffee"; // Discriminant field
  roast: "dark" | "medium" | "light";
  name: string;
};

type HotDrink = Tea | Coffee;

function drink(drink: HotDrink) {
  if (drink.type === "coffee") {
    drink.roast; // No more type error!
  }
}
```

Set Logic

```
type HotDrink = Tea | Coffee;
type OrderStyle = "forHere" | "toGo";

type DrinkOrder = ( Tea | Coffee ) & OrderStyle;
type AltDrinkOrder = ( Tea & OrderStyle ) | ( Coffee & OrderStyle )
```

- Intersections and unions operate just like logical & and |.
- We can chain them together just like we'd describe a set.