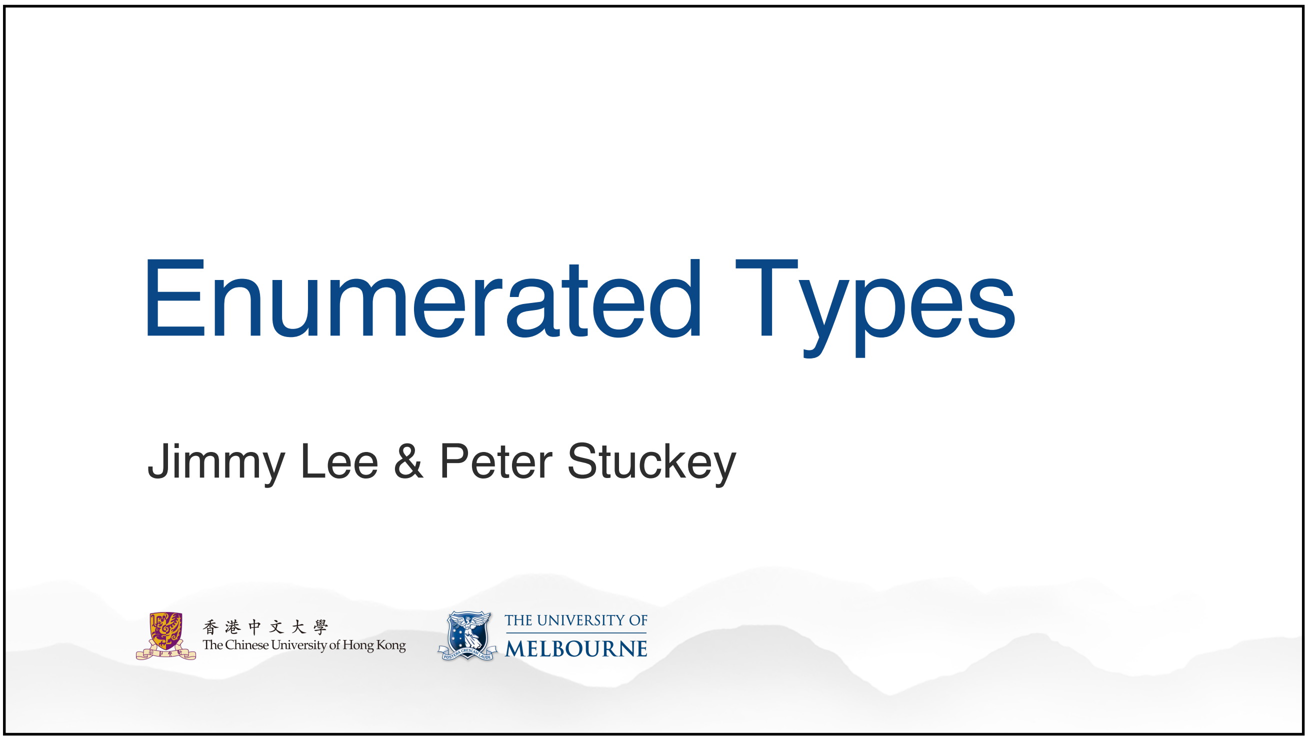
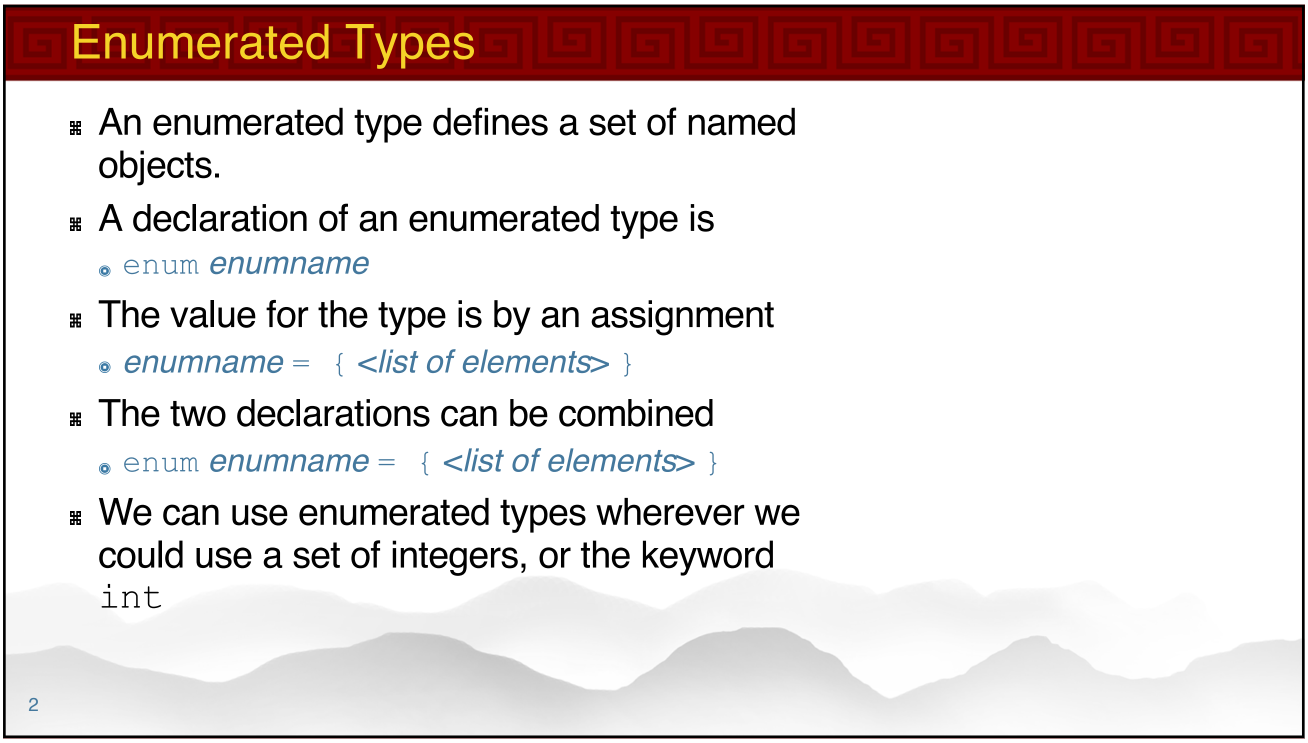


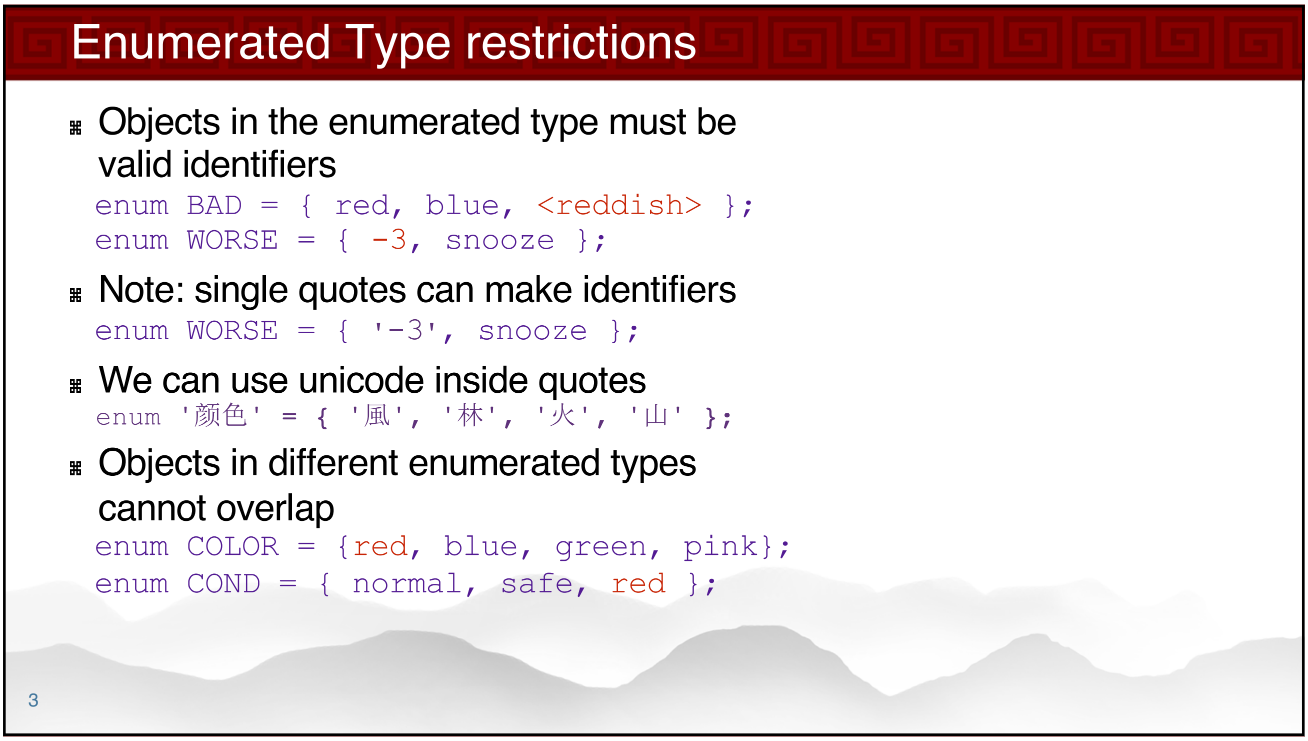
Enumerated Types

Jimmy Lee & Peter Stuckey



Enumerated Types

- An enumerated type defines a set of named objects.
- A declaration of an enumerated type is
  - `enum` *enumname*
- The value for the type is by an assignment
  - *enumname* = { *<list of elements>* }
- The two declarations can be combined
  - `enum` *enumname* = { *<list of elements>* }
- We can use enumerated types wherever we could use a set of integers, or the keyword `int`



Enumerated Type restrictions

- Objects in the enumerated type must be valid identifiers

```
enum BAD = { red, blue, <reddish> };
enum WORSE = { -3, snooze };
```

- Note: single quotes can make identifiers

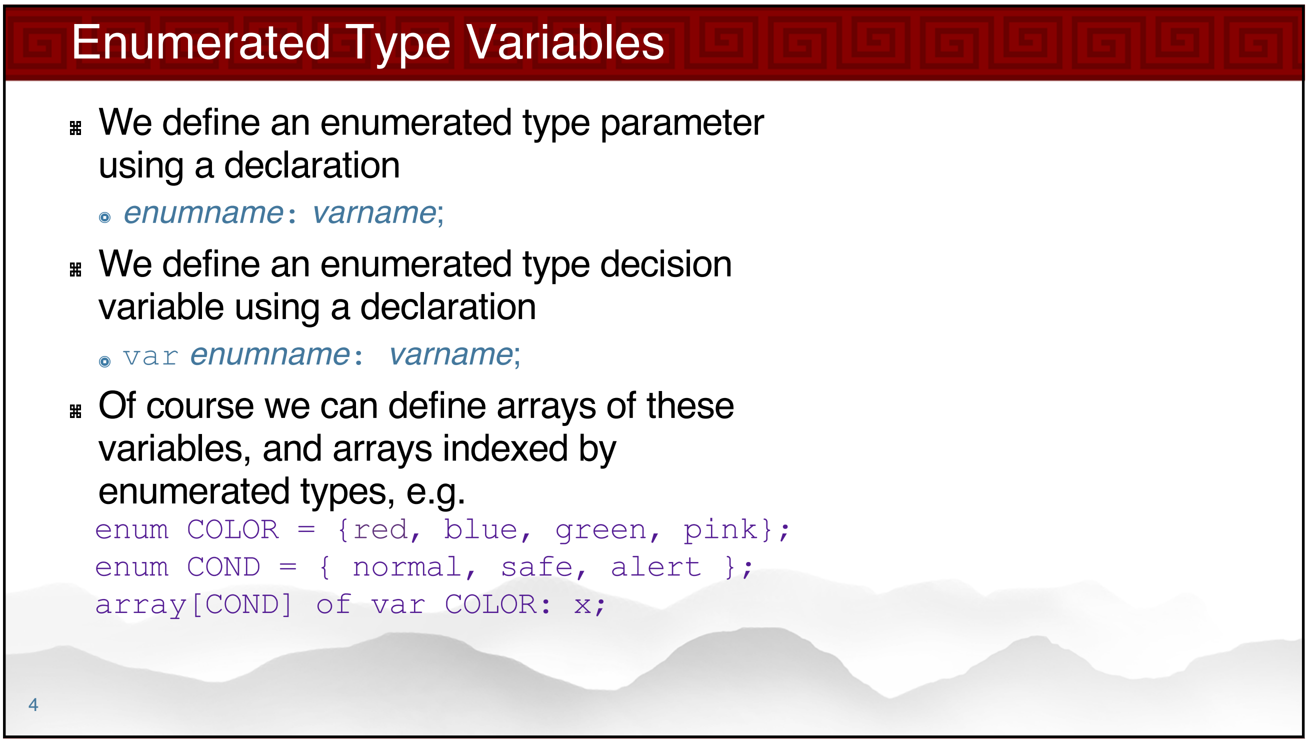
```
enum WORSE = { '-3', snooze };
```

- We can use unicode inside quotes

```
enum '颜色' = { '風', '林', '火', '山' };
```

- Objects in different enumerated types cannot overlap

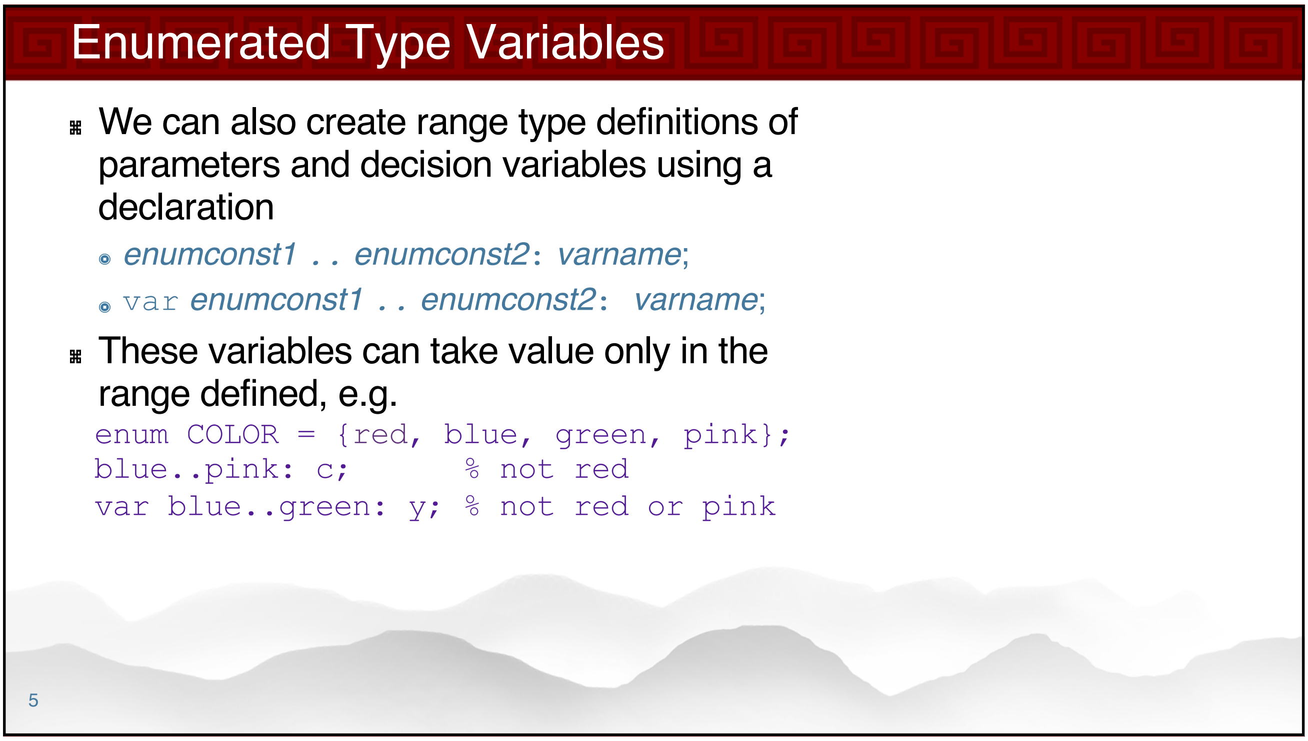
```
enum COLOR = {red, blue, green, pink};
enum COND = { normal, safe, red };
```



Enumerated Type Variables

- We define an enumerated type parameter using a declaration
  - *enumname*: *varname*;
- We define an enumerated type decision variable using a declaration
  - var *enumname*: *varname*;
- Of course we can define arrays of these variables, and arrays indexed by enumerated types, e.g.

```
enum COLOR = {red, blue, green, pink};
enum COND = { normal, safe, alert };
array[COND] of var COLOR: x;
```

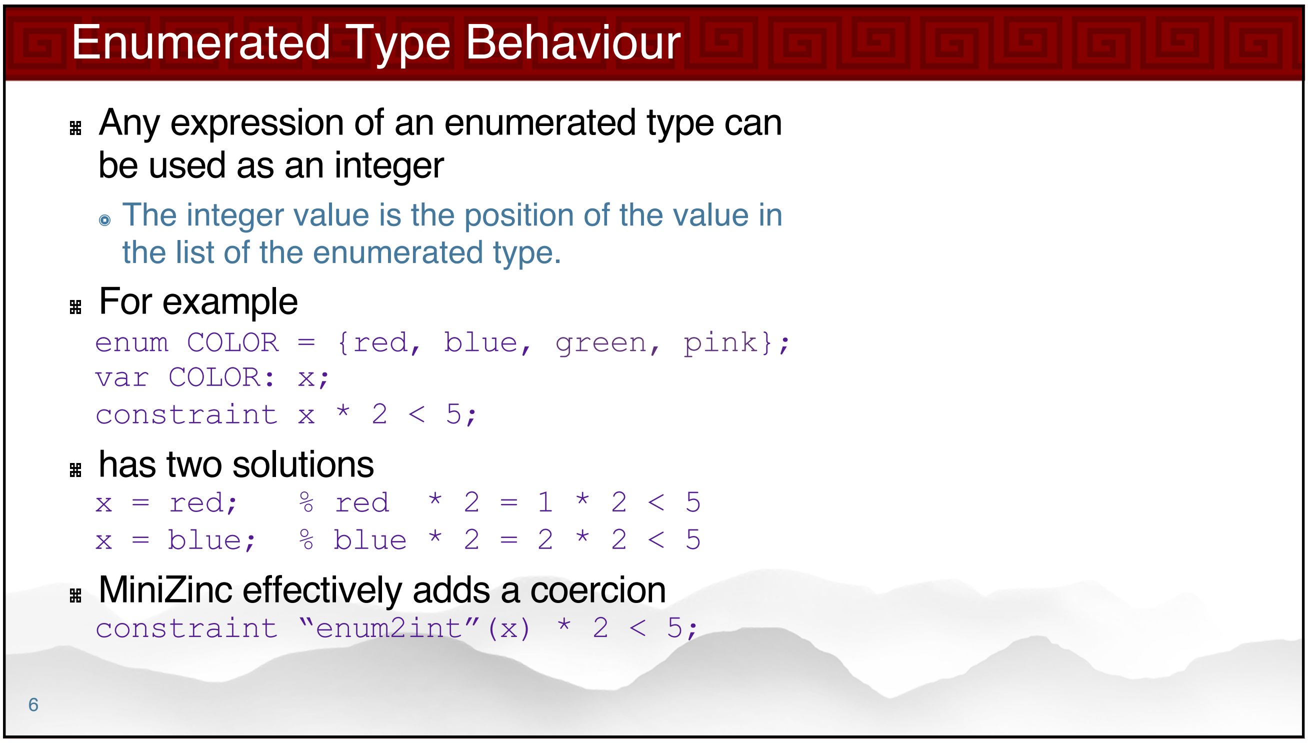


Enumerated Type Variables

- We can also create range type definitions of parameters and decision variables using a declaration
  - *enumconst1* . . *enumconst2*: *varname*;
  - `var` *enumconst1* . . *enumconst2*: *varname*;
- These variables can take value only in the range defined, e.g.

```
enum COLOR = {red, blue, green, pink};
blue..pink: c;       % not red
var blue..green: y; % not red or pink
```

5



Enumerated Type Behaviour

- Any expression of an enumerated type can be used as an integer
  - The integer value is the position of the value in the list of the enumerated type.
- For example

```
enum COLOR = {red, blue, green, pink};
var COLOR: x;
constraint x * 2 < 5;
```

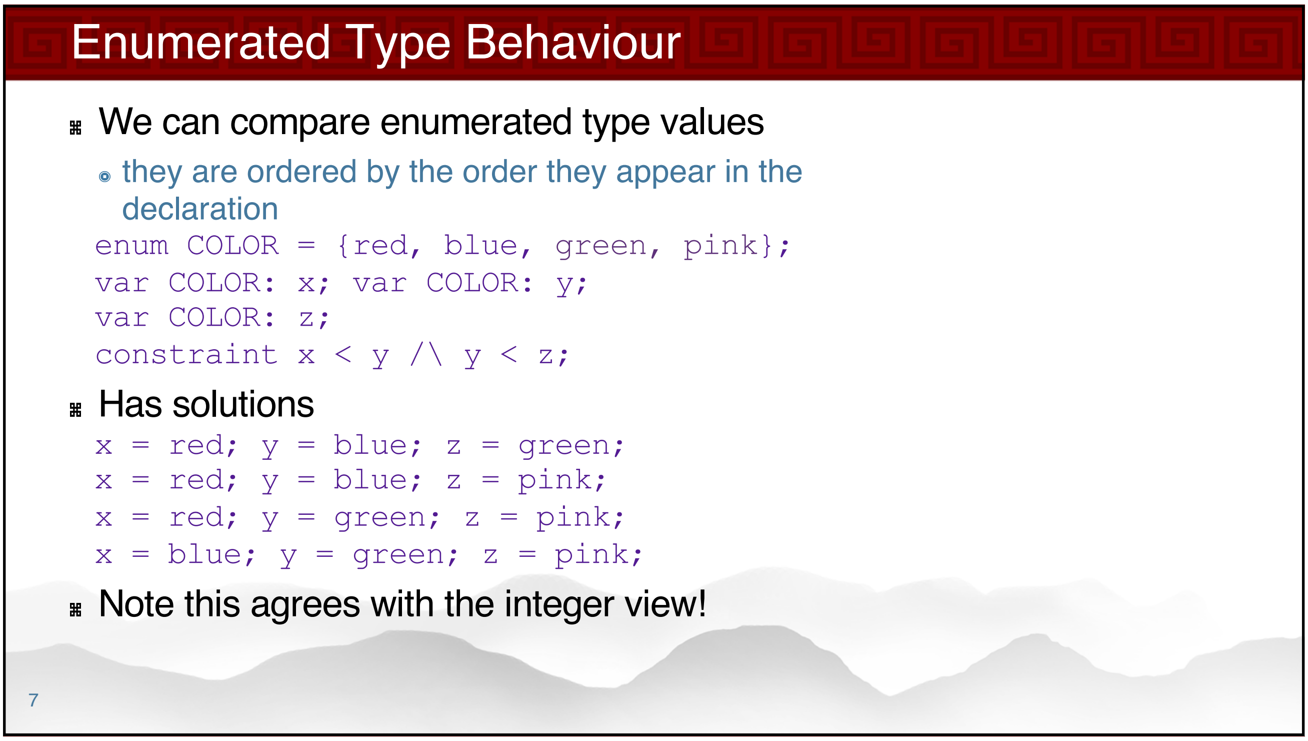
- has two solutions

```
x = red;   % red  * 2 = 1 * 2 < 5
x = blue;  % blue * 2 = 2 * 2 < 5
```

- MiniZinc effectively adds a coercion

```
constraint “enum2int”(x) * 2 < 5;
```

6



Enumerated Type Behaviour

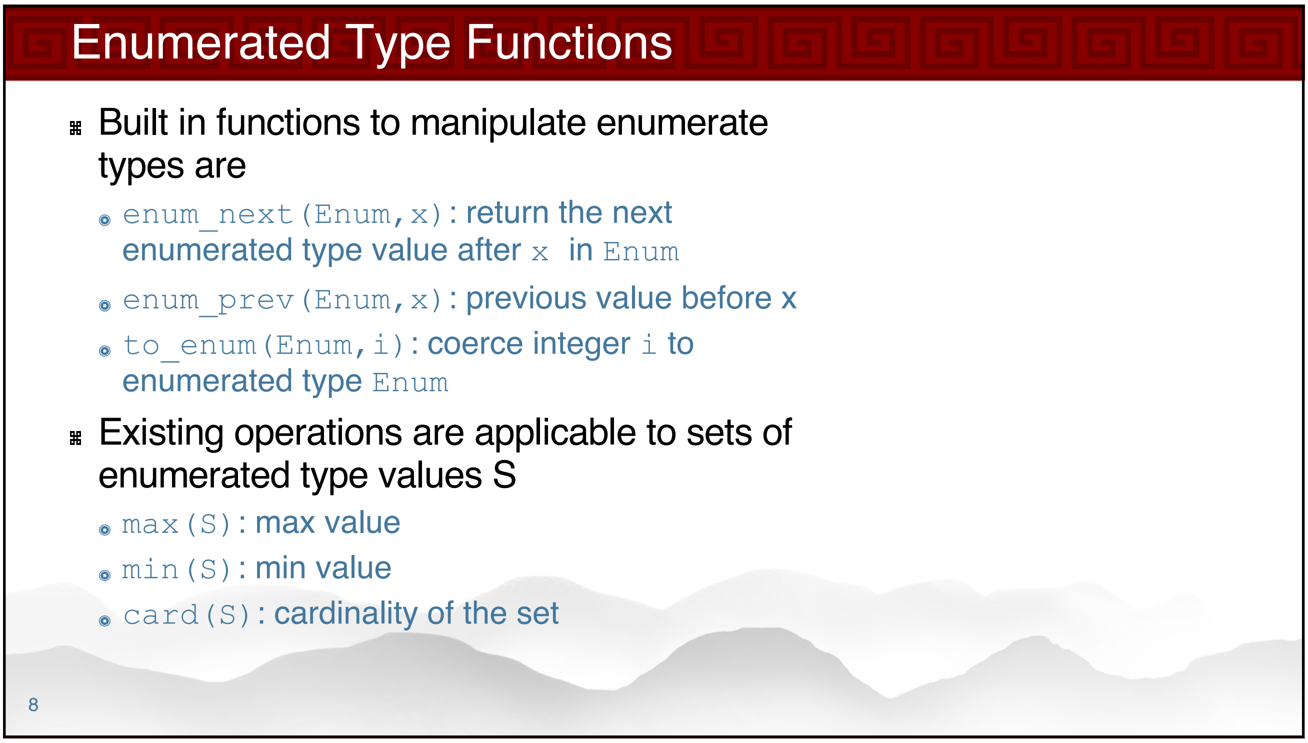
- We can compare enumerated type values
  - they are ordered by the order they appear in the declaration

```
enum COLOR = {red, blue, green, pink};
var COLOR: x; var COLOR: y;
var COLOR: z;
constraint x < y /\ y < z;
```

- Has solutions

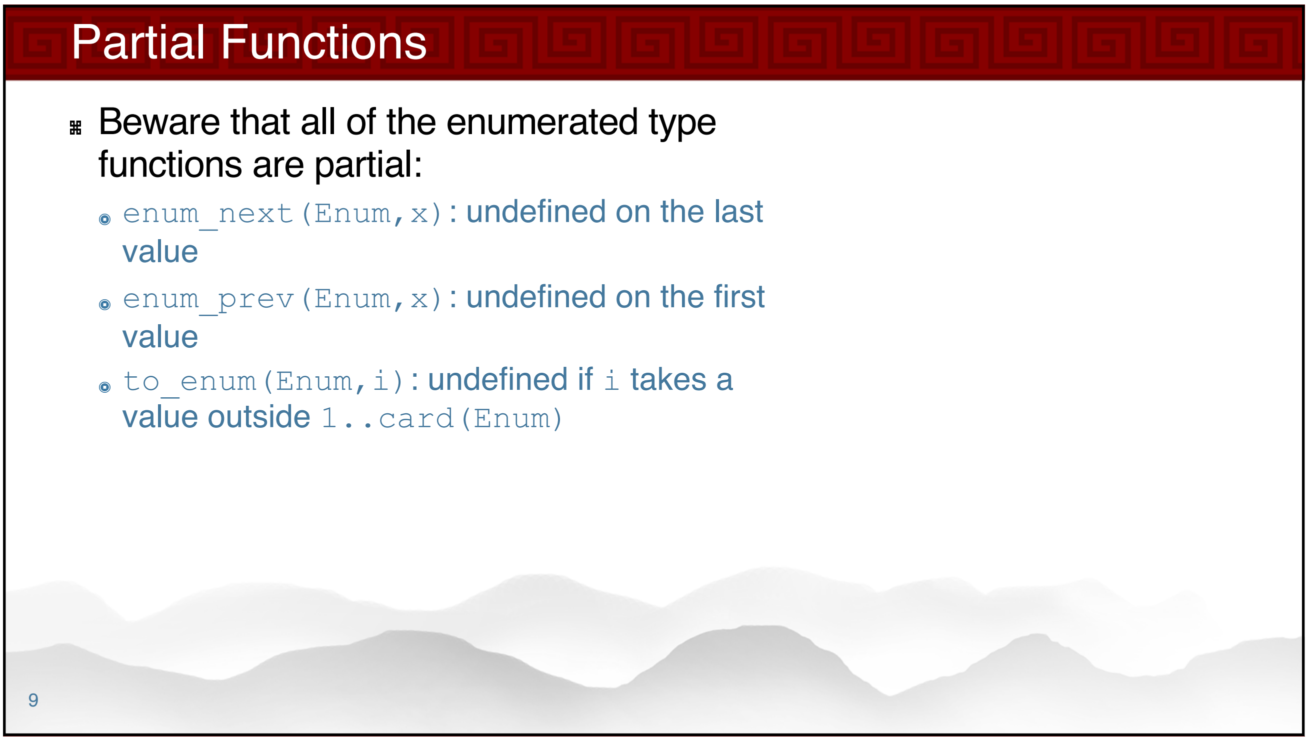
```
x = red; y = blue; z = green;
x = red; y = blue; z = pink;
x = red; y = green; z = pink;
x = blue; y = green; z = pink;
```

- Note this agrees with the integer view!



Enumerated Type Functions

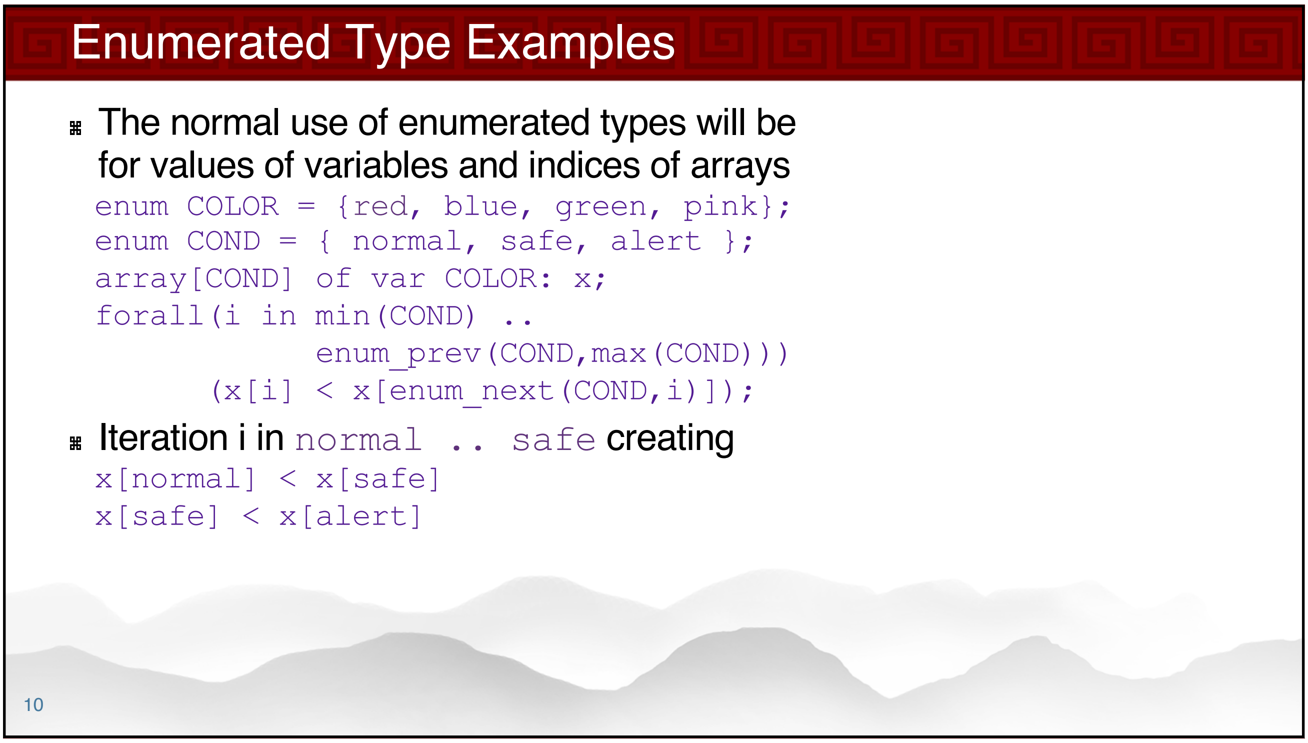
- Built in functions to manipulate enumerate types are
  - `enum_next(Enum,x)`: return the next enumerated type value after `x` in `Enum`
  - `enum_prev(Enum,x)`: previous value before x
  - `to_enum(Enum,i)`: coerce integer `i` to enumerated type `Enum`
- Existing operations are applicable to sets of enumerated type values S
  - `max(S)`: max value
  - `min(S)`: min value
  - `card(S)`: cardinality of the set



Partial Functions

- Beware that all of the enumerated type functions are partial:
  - `enum_next(Enum,x)`: undefined on the last value
  - `enum_prev(Enum,x)`: undefined on the first value
  - `to_enum(Enum,i)`: undefined if `i` takes a value outside `1..card(Enum)`

9



Enumerated Type Examples

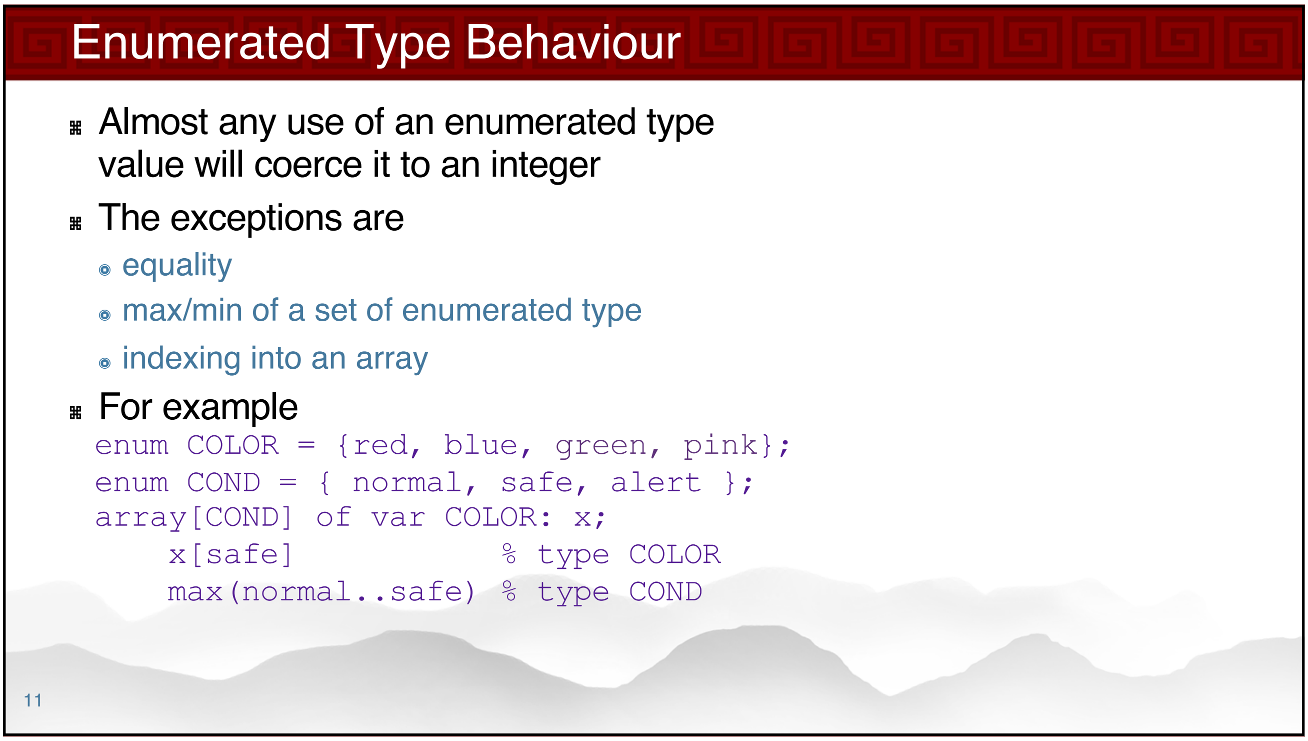
- The normal use of enumerated types will be for values of variables and indices of arrays

```
enum COLOR = {red, blue, green, pink};
enum COND = { normal, safe, alert };
array[COND] of var COLOR: x;
forall(i in min(COND) ..
            enum_prev(COND,max(COND)))
      (x[i] < x[enum_next(COND,i)]);
```

- Iteration i in `normal .. safe` creating

```
x[normal] < x[safe]
x[safe] < x[alert]
```

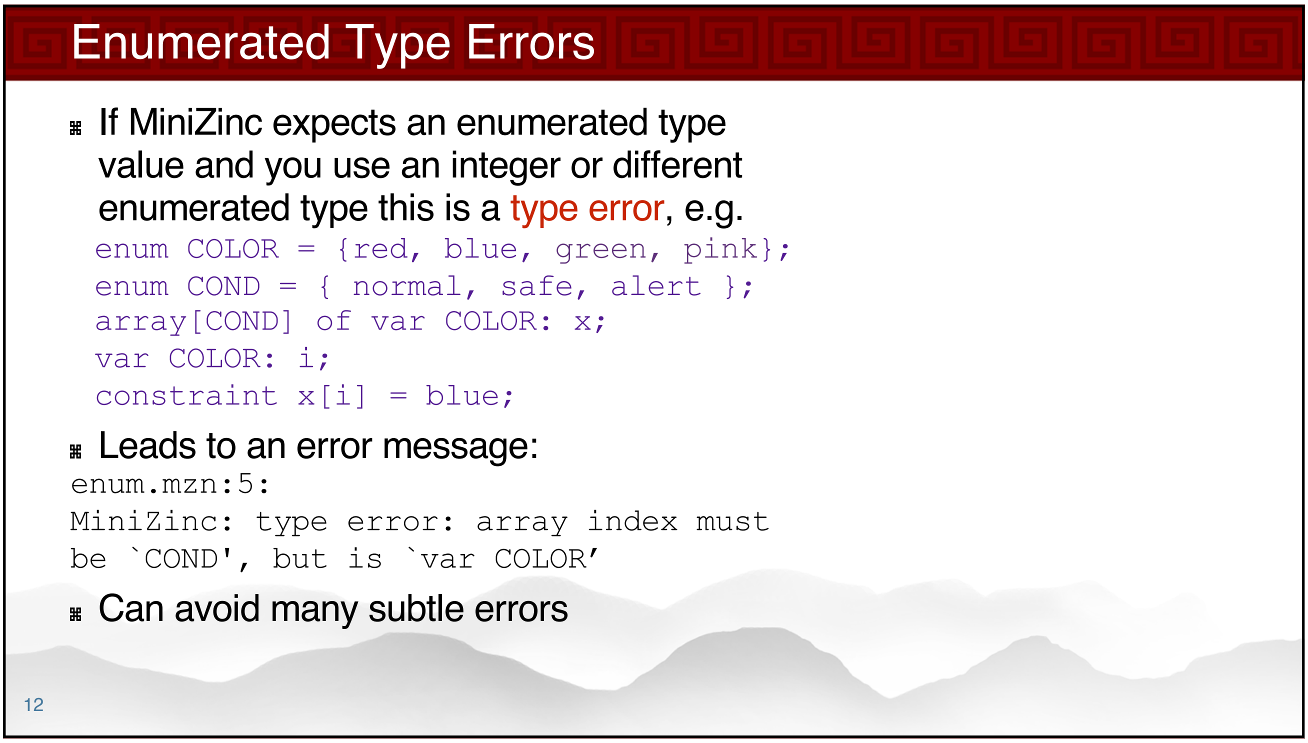
10



Enumerated Type Behaviour

- Almost any use of an enumerated type value will coerce it to an integer
- The exceptions are
  - equality
  - max/min of a set of enumerated type
  - indexing into an array
- For example

```
enum COLOR = {red, blue, green, pink};
enum COND = { normal, safe, alert };
array[COND] of var COLOR: x;
    x[safe]           % type COLOR
    max(normal..safe) % type COND
```



Enumerated Type Errors

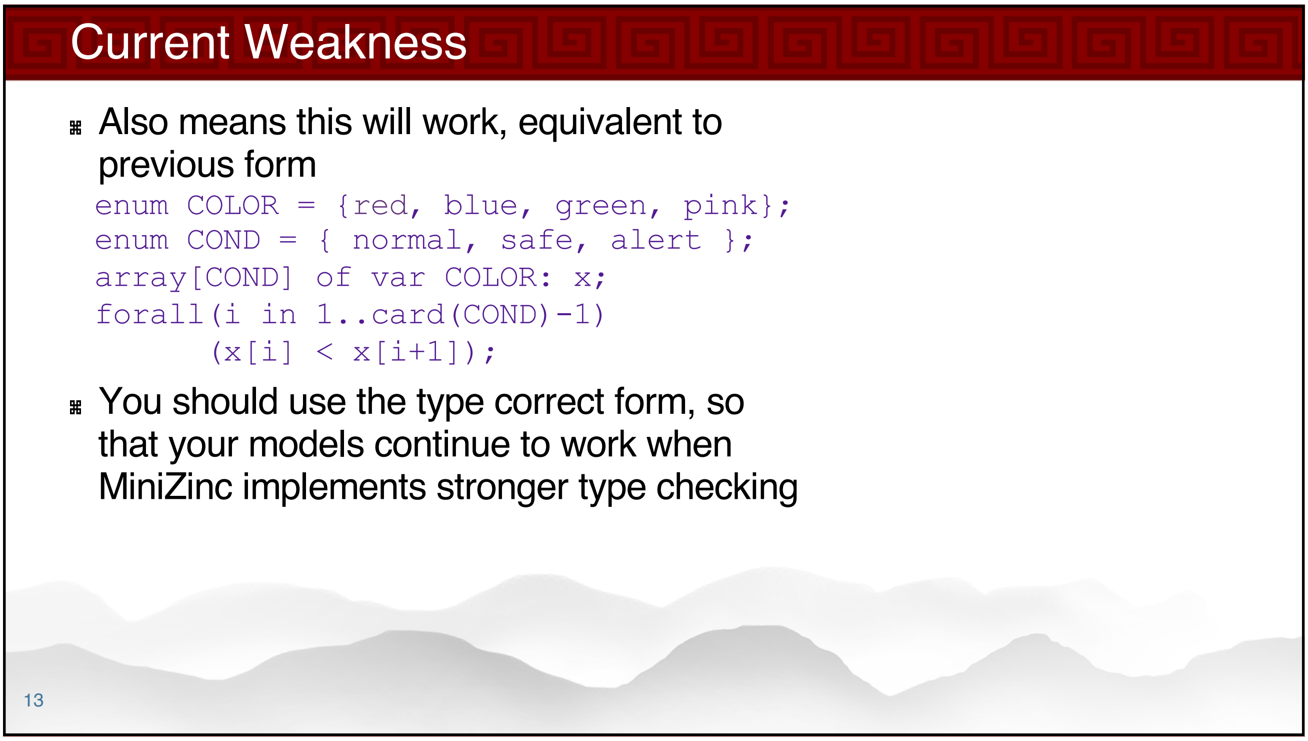
- If MiniZinc expects an enumerated type value and you use an integer or different enumerated type this is a type error, e.g.

```
enum COLOR = {red, blue, green, pink};
enum COND = { normal, safe, alert };
array[COND] of var COLOR: x;
var COLOR: i;
constraint x[i] = blue;
```

- Leads to an error message:

```
enum.mzn:5:
MiniZinc: type error: array index must
be `COND', but is `var COLOR’
```

- Can avoid many subtle errors

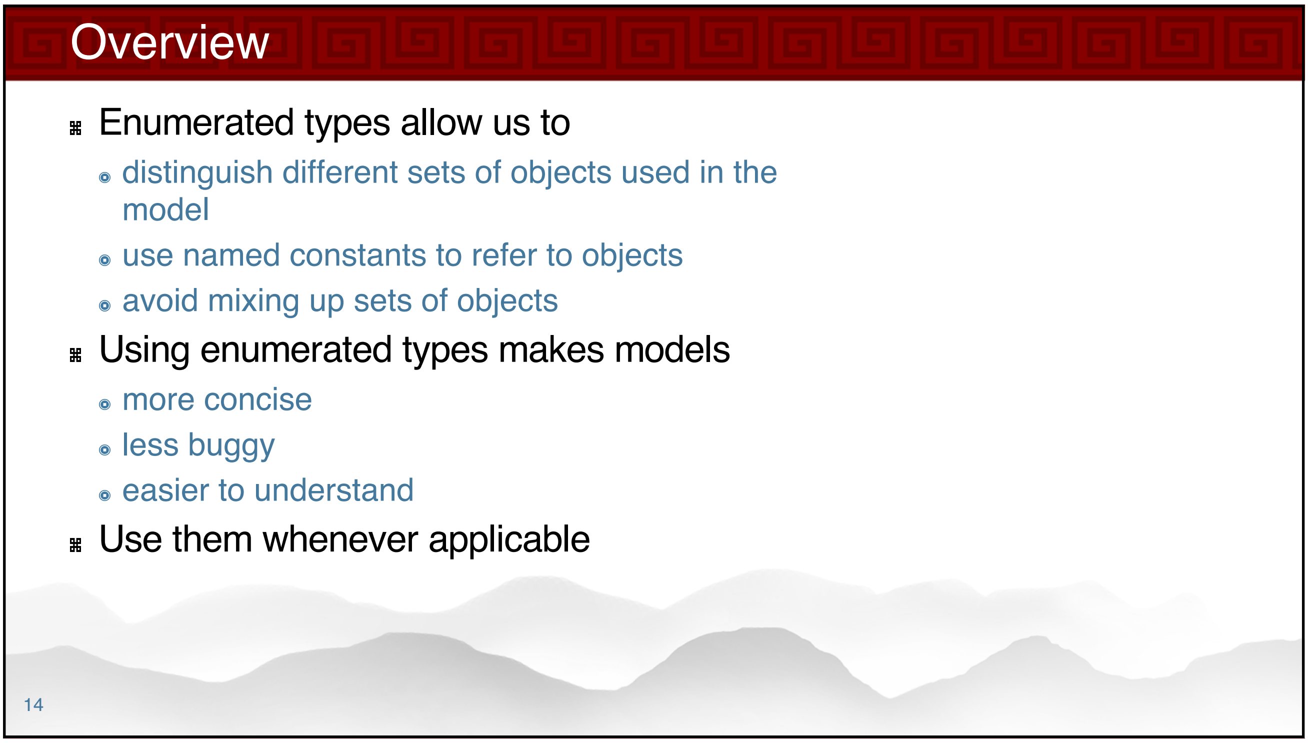


Current Weakness

- Also means this will work, equivalent to previous form

```
enum COLOR = {red, blue, green, pink};
enum COND = { normal, safe, alert };
array[COND] of var COLOR: x;
forall(i in 1..card(COND)-1)
      (x[i] < x[i+1]);
```

- You should use the type correct form, so that your models continue to work when MiniZinc implements stronger type checking



Overview

- Enumerated types allow us to
  - distinguish different sets of objects used in the model
  - use named constants to refer to objects
  - avoid mixing up sets of objects
- Using enumerated types makes models
  - more concise
  - less buggy
  - easier to understand
- Use them whenever applicable