Polymorphism

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- Beyond type equality
- Defining our own subtypes
- Lists and state machines

Beyond type equality

Beyond type equality

Introduction

- In the previous lecture we have seen that Java/C# have a static type system
- Programs that make no sense are outright refused by the compiler
- "Make no sense" means calling methods or making assignments with the wrong types

Beyond type equality

So far, we defined a type to be wrong if it is not exactly the same as what we expected.

If we expected an `int`, but got a `Person`, then clearly something was off and we expect[a] a compiler error.

[a] welcome, actually

Beyond type equality

Wrong types

The two most important typing rules for these violations are those of variable assignment and method call.

Beyond type equality

When assigning a variable, we expect the type of the expression and that of the variable to be exactly the same: $T = D[x]$

$$\frac{\langle \mathtt{e}, D\rangle \rightarrow \langle T, D\rangle \wedge \ T = D[x]}{\langle \mathtt{x} = \mathtt{e}, D\rangle \rightarrow \langle D[\mathtt{void}], D\rangle}$$

Beyond type equality

When calling a method, we expect the type of the parameters and those of the arguments to be exactly the same: $\mathtt{P}_i = \mathtt{P}'_i$

$$\frac{\langle \mathtt{c}, D\rangle \rightarrow \langle \mathtt{C}, D\rangle \wedge \langle \mathtt{m}, C\rangle \rightarrow \langle (\mathtt{P}_1 \times \mathtt{P}_2 \times \cdots \times \mathtt{P}_n \rightarrow \mathtt{R}), C\rangle \wedge \langle \mathtt{p_i}, D\rangle \rightarrow \langle \mathtt{P}'_i, D\rangle \wedge \mathtt{P}_i = \mathtt{P}'_i}{\langle (\mathtt{c.m}(..\mathtt{p_i}..)), D\rangle \rightarrow \langle \mathtt{R}, D\rangle}$$

Beyond type equality

For example, the program below violates typing rules:

```
int x = 5;
x = "uh!?";
```

Beyond type equality

For example, the program below violates typing rules:

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void Incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
Counter c = new Counter();
c.Incr(c);
```

Beyond type equality

`int x = 5.5;` makes no sense

This will also give a compiler error.

Beyond type equality

Always wrong?

- What about a similar expression, such as `float x = 10;`?
- `x` is a `float`
- `10` is an `int`
- `float` $\neq$ `int`, so we should get a type error.

Beyond type equality

`float x = 10;` makes sense, even though it violates the strictest typing rules described so far.

Floating point numbers "contain" integers, so converting the integer 10 to 10.0 loses no information

Subtyping

This means that the typing rules as seen so far are **too restrictive**: we should be able to accept an assignment from a more specific data type (such as int) to a less specific data type (such as `float`).

Beyond type equality

Consider any two data types, `T` and `S`

We say that `S <: T`, read "S is a subtype of T", to mean that any value of type `S` can be safely used where a value of type `T` is expected.

We also say that, when `S <: T`, `T` **generalizes** `S`.

Beyond type equality

More and less specific data types

For example, `int <: float`, because any value of type `int` can be safely used where a value of type `float` is expected (as the conversion loses no data, and thus can be inserted by the compiler itself).

Beyond type equality

More and less specific data types

We can now amend many of our typing rules: instead of type equality, we can use subtyping to preserve safety, but achieve more flexibility.

Beyond type equality

When assigning a variable, we expect the type of the expression **to be a subtype** of that of the variable.

$$\frac{\langle \mathtt{e}, D\rangle \rightarrow \langle T, D\rangle \wedge \ T <: D[x]}{\langle \mathtt{x} = \mathtt{e}, D\rangle \rightarrow \langle D[\mathtt{void}], D\rangle}$$

Beyond type equality

When calling a method, we expect the type of the parameters **to be subtypes** of the types of the arguments.

$$\frac{\langle \texttt{c}, D\rangle \rightarrow \langle \texttt{C}, D\rangle \wedge \langle \texttt{m}, C\rangle \rightarrow \langle (\texttt{P}_1 \times \texttt{P}_2 \times \cdots \times \texttt{P}_n \rightarrow \texttt{R}), C\rangle \wedge \langle \texttt{p}_\texttt{i}, D\rangle \rightarrow \langle \texttt{P}'_i, D\rangle \wedge \texttt{P}_i :> \texttt{P}'_i}{\langle (\texttt{c.m}(..\texttt{p}_\texttt{i}..)), D\rangle \rightarrow \langle \texttt{R}, D\rangle}$$

Defining our own subtypes

HOGESCHOOL ROTTERDAM

Defining our own subtypes

Remember that we can define our own classes to extend the data types of the language when they are insufficient for our domain.

We can make use of subtyping for those custom classes.

This makes it possible to capture **generalization relationships** between our own data types.

Defining our own subtypes

Can you think of a few examples of classes that generalize each other?

Defining our own subtypes

Can you think of a few examples of classes that generalize each other?

`Person` generalizes `Student`.

`LightEmitter` generalizes `Lamp`.

`Animal` generalizes `Dog`.

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`?

Defining our own subtypes

Which generalizes which?

`Mercedes vs CarBrand? CarBrand :> Mercedes`

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`?

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`?

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`?

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`? `Person :> Employee`

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`? `Person :> Employee`

`Student` vs `Person`?

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`? `Person :> Employee`

`Student` vs `Person`? `Person :> Student`.

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`? `Person :> Employee`

`Student` vs `Person`? `Person :> Student.`

`Student` vs `Employee`?

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`? `Person :> Employee`

`Student` vs `Person`? `Person :> Student`.

`Student` vs `Employee`? Neither.

Defining our own subtypes

Which generalizes which?

`Mercedes` vs `CarBrand`? `CarBrand :> Mercedes`

`LivingSpace` vs `Apartment`? `LivingSpace :> Apartment`

`Cat` vs `Bird`? Neither.

`Person` vs `Employee`? `Person :> Employee`

`Student` vs `Person`? `Person :> Student`.

`Student` vs `Employee`? Neither.

Defining our own subtypes

Consider, in particular, the following relationships:

```
Person :> Employee
```

```
Person :> Student
```

Defining our own subtypes

Consider, in particular, the following relationships:

`Person :> Employee`

`Person :> Student`

We could imagine that `Person`, `Employee`, and `Student` are all classes

Moreover, `Person` and `Employee` are somehow related

Similarly, `Person` and `Student` are related in the same way

Defining our own subtypes

Class inheritance

- This relationships is modeled, in Java/C#, with class inheritance.
- A class such as `Employee` will therefore inherit from class `Person`
- This means that `Employee` will automatically have all fields and methods of `Person`

Defining our own subtypes

When we use inheritance, for example of `Person` from `Employee`, then the language automatically infers that `Person :> Employee`

This means therefore that anywhere in the language (a variable, a parameter, etc.) where we expected a `Person`, we can give an `Employee`.

This provides polymorphism, as the same data type (in this case `Person`) can have multiple shapes: a `Person`, an `Employee`, a `Student`, etc.

Defining our own subtypes

Inheritance requires very little code: in C#, a colon (:) suffices with the name of the inherited class next to that of the defined class.

In Java, we use the keyword `extends` instead of the colon.

It is possible to inherit at most one class.

Defining our own subtypes

The program below works: why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Defining our own subtypes

The program below works: why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Because the declaration of `b` specifies `A` as the type, but whenever we expect an `A` we can use a `B` thanks to inheritance.

Defining our own subtypes

Which in Java then becomes:

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B extends A {
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
System.out.println(b.M(5));
```

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Declarations:

PC
1

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Declarations:

PC
7

Classes:

A
A=A $\rightarrow$ A M=(A$\times$int) $\rightarrow$ int

Defining our own subtypes

```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Declarations:

PC
14

Classes:

A	B
$A = A \to A$ $M = (A \times int) \to int$	$B = B \to B$ $M = (A \times int) \to int$ $N = (B \times int) \to int$

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Declarations:

PC	b
16	A

Classes:

A	B
A=A $\rightarrow$ A M=(A$\times$int) $\rightarrow$ int	B=B $\rightarrow$ B M=(A$\times$int) $\rightarrow$ int N=(B$\times$int) $\rightarrow$ int

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Declarations:

b		PC	ret	arg_1	this
A		16	null	int	A

Classes:

A	B
A=A $\rightarrow$ A M=(A$\times$int) $\rightarrow$ int	B=B $\rightarrow$ B M=(A$\times$int) $\rightarrow$ int N=(B$\times$int) $\rightarrow$ int

Defining our own subtypes


```
1  class A {
2    public A() {
3    }
4    public int M(int x) {
5      return (x + x);
6    }
7  }
8  class B : A {
9    public B() {
10   }
11   public int N(int y) {
12     return (y * 10);
13   }
14 }
15 A b = new B();
16 Console.WriteLine(b.M(5));
```

Declarations:

b		PC	ret	arg_1	this
A		16	int	int	A

Classes:

A	B
A=A $\rightarrow$ A M=(A$\times$int) $\rightarrow$ int	B=B $\rightarrow$ B M=(A$\times$int) $\rightarrow$ int N=(B$\times$int) $\rightarrow$ int

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Declarations:

PC	b
17	A

Classes:

A	B
$A = A \to A$ $M = (A \times int) \to int$	$B = B \to B$ $M = (A \times int) \to int$ $N = (B \times int) \to int$

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC
1

Defining our own subtypes

```
1  class A {
2    public A() {
3    }
4    public int M(int x) {
5      return (x + x);
6    }
7  }
8  class B : A {
9    public B() {
10   }
11   public int N(int y) {
12     return (y * 10);
13   }
14 }
15 A b = new B();
16 Console.WriteLine(b.M(5));
```

Stack:

PC
15

Heap:

1

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC	...		PC	ret	this
15	...		10	null	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC	...		PC	ret
15	...		10	ref 1

Heap:

1

HOGESCHOOL ROTTERDAM

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC	b
16	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC	...		PC	ret	this	x
16	...		5	null	ref 1	5

Heap:

1

```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC	...		PC	ret
16	...		5	10

Heap:

1

Defining our own subtypes


```
class A {
  public A() {
  }
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public B() {
  }
  public int N(int y) {
    return (y * 10);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Stack:

PC	b
17	ref 1

Heap:

1

Output: 10

Defining our own subtypes

Class inheritance

Similarly, given a method that expects a parameter of type `A` could accept a parameter of type `C`.

Defining our own subtypes

Inheritance does not make all combinations possible. For example, the program below does not work: why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : A {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
```

Defining our own subtypes

Inheritance does not make all combinations possible. For example, the program below does not work: why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : A {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
```

There is no inheritance relationship between `B` and `C`!

Which in Java then becomes:

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B extends A {
  public int N(int y) {
    return (y * 10);
  }
}
class C extends A {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
System.out.println(b.M(5));
```

Defining our own subtypes

Class inheritance

The subtyping relationship is transitive. This means that given X ¡: Y and Y ¡: Z, implies X ¡: Z.

Defining our own subtypes

Inheritance can also perform multiple conversion steps:

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Defining our own subtypes

Which in Java then becomes:

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B extends A {
  public int N(int y) {
    return (y * 10);
  }
}
class C extends B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
System.out.println(a.M(5));
```

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

PC
1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

PC
5

Classes:

A
M=(A×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

PC
10

Classes:

A	B
M=(A×int) → int	M=(A×int) → int N=(B×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

PC
15

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
1  class A {
2    public int M(int x) {
3      return (x + x);
4    }
5  }
6  class B : A {
7    public int N(int y) {
8      return (y * 10);
9    }
10 }
11 class C : B {
12   public int O(int z) {
13     return (z - 1);
14   }
15 }
16 A a = new C();
17 Console.WriteLine(a.M(5));
```

Declarations:

PC	a
17	A

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

a		PC	ret	arg_1	this
A		17	null	int	A

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

a		PC	ret	arg_1	this
A		17	int	int	A

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Declarations:

PC	a
18	A

Classes:

A	B	C
M=(A×int) $\rightarrow$ int	M=(A×int) $\rightarrow$ int N=(B×int) $\rightarrow$ int	M=(A×int) $\rightarrow$ int N=(B×int) $\rightarrow$ int O=(C×int) $\rightarrow$ int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC
1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC
18

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC	...		PC	ret	this
18	...		13	null	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC	...		PC	ret
18	...		13	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC	a
19	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC	...		PC	ret	this	x
19	...		3	null	ref 1	5

Heap:

1

Defining our own subtypes


```
1  class A {
2    public int M(int x) {
3      return (x + x);
4    }
5  }
6  class B : A {
7    public int N(int y) {
8      return (y * 10);
9    }
10 }
11 class C : B {
12   public C() {
13   }
14   public int O(int z) {
15     return (z - 1);
16   }
17 }
18 A a = new C();
19 Console.WriteLine(a.M(5));
```

Stack:

PC	...		PC	ret
19	...		3	10

Heap:

1

Defining our own subtypes

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
A a = new C();
Console.WriteLine(a.M(5));
```

Stack:

PC	a
20	ref 1

Heap:

1

Output: 10

Defining our own subtypes

Of course, when inheriting, we can still use all methods available given a variable type.

The code below does indeed work. Why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Defining our own subtypes

Of course, when inheriting, we can still use all methods available given a variable type.

The code below does indeed work. Why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

It is possible to call both methods `M` and `N` on an instance of `B`.

Which in Java then becomes:

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B extends A {
  public int N(int y) {
    return (y * 10);
  }
}
class C extends B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
System.out.println(b.M(5));
System.out.println(b.N(5));
```

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

PC
1

Defining our own subtypes

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

PC
5

Classes:

A
M=(A $\times$ int) $\rightarrow$ int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

PC
10

Classes:

A	B
M=(A×int) → int	M=(A×int) → int N=(B×int) → int

Defining our own subtypes

```
1  class A {
2    public int M(int x) {
3      return (x + x);
4    }
5  }
6  class B : A {
7    public int N(int y) {
8      return (y * 10);
9    }
10 }
11 class C : B {
12   public int O(int z) {
13     return (z - 1);
14   }
15 }
16 B b = new C();
17 Console.WriteLine(b.M(5));
18 Console.WriteLine(b.N(5));
```

Declarations:

PC
15

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

PC	b
17	B

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

b		PC	ret	arg_1	this
B		17	null	int	B

Classes:

A	B	C
$M=(A \times int) \rightarrow int$	$M=(A \times int) \rightarrow int$ $N=(B \times int) \rightarrow int$	$M=(A \times int) \rightarrow int$ $N=(B \times int) \rightarrow int$ $O=(C \times int) \rightarrow int$

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

b		PC	ret	arg_1	this
B		17	int	int	B

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

PC	b
18	B

Classes:

A	B	C
$M = (A \times int) \rightarrow int$	$M = (A \times int) \rightarrow int$ $N = (B \times int) \rightarrow int$	$M = (A \times int) \rightarrow int$ $N = (B \times int) \rightarrow int$ $O = (C \times int) \rightarrow int$

Defining our own subtypes

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

b		PC	ret	arg_1	this
B		18	null	int	B

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

b		PC	ret	arg_1	this
B		18	int	int	B

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Declarations:

PC	b
19	B

Classes:

A	B	C
M=(A×int) → int	M=(A×int) → int N=(B×int) → int	M=(A×int) → int N=(B×int) → int O=(C×int) → int

Defining our own subtypes

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC
1

Defining our own subtypes


```
1  class A {
2    public int M(int x) {
3      return (x + x);
4    }
5  }
6  class B : A {
7    public int N(int y) {
8      return (y * 10);
9    }
10 }
11 class C : B {
12   public C() {
13   }
14   public int O(int z) {
15     return (z - 1);
16   }
17 }
18 B b = new C();
19 Console.WriteLine(b.M(5));
20 Console.WriteLine(b.N(5));
```

Stack:

PC
18

Heap:

1

Defining our own subtypes


```
1  class A {
2    public int M(int x) {
3      return (x + x);
4    }
5  }
6  class B : A {
7    public int N(int y) {
8      return (y * 10);
9    }
10 }
11 class C : B {
12   public C() {
13   }
14   public int O(int z) {
15     return (z - 1);
16   }
17 }
18 B b = new C();
19 Console.WriteLine(b.M(5));
20 Console.WriteLine(b.N(5));
```

Stack:

PC	...		PC	ret	this
18	...		13	null	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	...		PC	ret
18	...		13	ref 1

Heap:

1

Defining our own subtypes

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	b
19	ref 1

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	...		PC	ret	this	x
19	...		3	null	ref 1	5

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	...		PC	ret
19	...		3	10

Heap:

1

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	b
20	ref 1

Heap:

1

Output: 10

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	...		PC	ret	this	y
20	...		8	null	ref 1	5

Heap:

1

Output: 10

Defining our own subtypes


```
1  class A {
2    public int M(int x) {
3      return (x + x);
4    }
5  }
6  class B : A {
7    public int N(int y) {
8      return (y * 10);
9    }
10 }
11 class C : B {
12   public C() {
13   }
14   public int O(int z) {
15     return (z - 1);
16   }
17 }
18 B b = new C();
19 Console.WriteLine(b.M(5));
20 Console.WriteLine(b.N(5));
```

Stack:

PC	...		PC	ret
20	...		8	50

Heap:

1

Output: 10

Defining our own subtypes


```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public C() {
  }
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.M(5));
Console.WriteLine(b.N(5));
```

Stack:

PC	b
21	ref 1

Heap:

1

Output:

10	50

Defining our own subtypes

A major difference with Python is that, even if the instance may allow calling some methods, subtyping might disallow it.

The code below does not work. Why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.O(5));
```

Defining our own subtypes

A major difference with Python is that, even if the instance may allow calling some methods, subtyping might disallow it.

The code below does not work. Why?

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B : A {
  public int N(int y) {
    return (y * 10);
  }
}
class C : B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
Console.WriteLine(b.O(5));
```

b is declared with type B, which has no method O.

Which in Java then becomes:

```
class A {
  public int M(int x) {
    return (x + x);
  }
}
class B extends A {
  public int N(int y) {
    return (y * 10);
  }
}
class C extends B {
  public int O(int z) {
    return (z - 1);
  }
}
B b = new C();
System.out.println(b.O(5));
```

Live code demo: person, employee, and student/animal, dog, cat/...

Interfaces and polymorphism

Interfaces and polymorphism

Zero-level data types

- The subtyping relationship can start from a data type so general that it has no concrete implementation.
- This data type is called an **interface**.
- Interfaces are classes defined with the keyword `interface`. They have no fields, and no implementation of their methods.
- We say that a class implements, not inherits from, one or more interfaces.

Interfaces and polymorphism

Zero-level data types

- Interfaces are especially useful to specify what requirements a data type must satisfy to be used in a context.
- With interfaces we are not bound to also giving a "default" implementation.

Interfaces and polymorphism

Implementing interfaces requires very little code: in C#, a colon (:) suffices with the name of the implemented interface next to that of the defined class.

In Java, we use the keyword `implements` instead of the colon.

It is possible to implement multiple interfaces from the same class.

The program below works: why?

```
interface A {
  int M(int x);
}
class B : A {
  public int M(int x) {
    return (x + x);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

The program below works: why?

```
interface A {
  int M(int x);
}
class B : A {
  public int M(int x) {
    return (x + x);
  }
}
A b = new B();
Console.WriteLine(b.M(5));
```

Because the declaration of `b` specifies `A` as the type, but whenever we expect an `A` we can use a `B` thanks to the subtyping of implemented interfaces.

Which in Java then becomes:

```
interface A {
  int M(int x);
}
class B implements A {
  public int M(int x) {
    return (x + x);
  }
}
A b = new B();
System.out.println(b.M(5));
```

The program below does not work: why?

```
interface A {
  int M(int x);
}
A a = new A();
Console.WriteLine(a.M(5));
```

The program below does not work: why?

```
interface A {
  int M(int x);
}
A a = new A();
Console.WriteLine(a.M(5));
```

Because `A` has no implementation and so cannot be instantiated: what code could we possibly execute for method `M`?

Interfaces and polymorphism

Which in Java then becomes:

```
interface A {
  int M(int x);
}
A a = new A();
System.out.println(a.M(5));
```

Interfaces and polymorphism

Zero-level data types

Polymorphism can be used in a lot of contexts, as long as the conversion we expect of the language is provably safe.

The program below works: why?

```
interface A {
  A M(A x);
}
class B : A {
  public A M(A x) {
    return this;
  }
}
A b = new B();
Console.WriteLine(b.M(new B()));
```

Interfaces and polymorphism

The program below works: why?

```
interface A {
  A M(A x);
}
class B : A {
  public A M(A x) {
    return this;
  }
}
A b = new B();
Console.WriteLine(b.M(new B()));
```

Because the argument to `M`, which should be an `A`, can safely accept a `B` as well.

Which in Java then becomes:

```
interface A {
  A M(A x);
}
class B implements A {
  public A M(A x) {
    return this;
  }
}
A b = new B();
System.out.println(b.M(new B()));
```

Live code demo: list, empty, and node.

Conclusion

Conclusion

Looking back

- Polymorphism makes it possible to pass different data types to other contexts, as long as the conversion is safe
- Inheritance is the basic mechanism of polymorphism
- Interfaces make this even more powerful by allowing the use of polymorphism without a concrete data type

This is it!

The best of luck, and thanks for the attention!