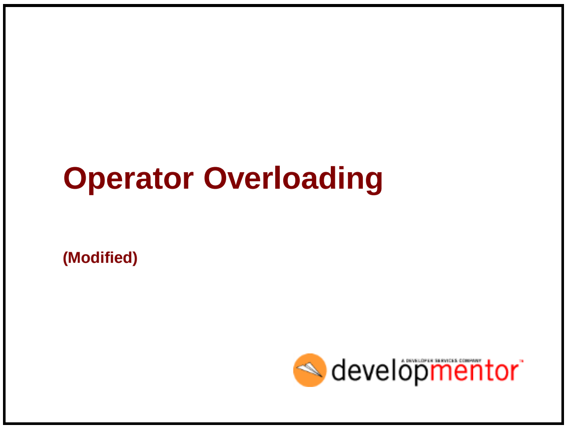
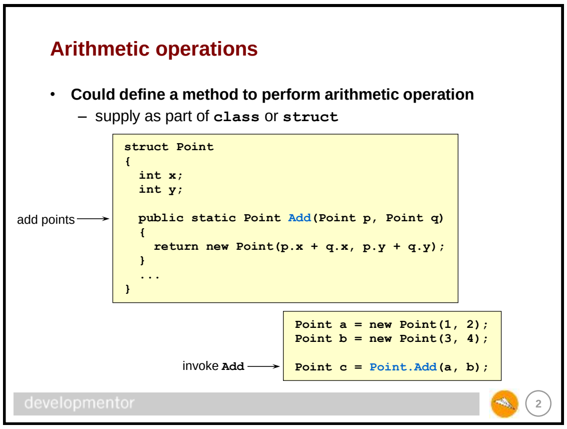
21: Operator Overloading

12/01/2003



Operator Overloading

(Modified)



Arithmetic operations

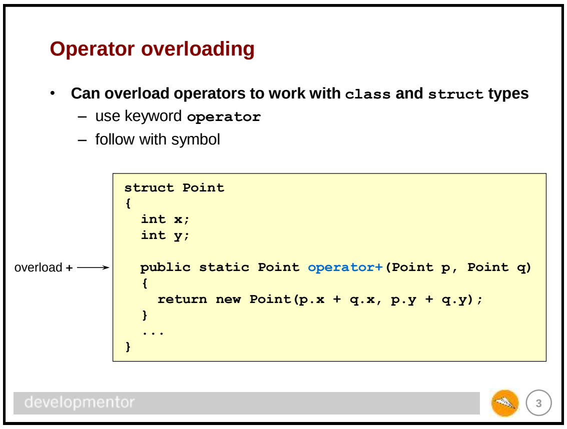
- **Could define a method to perform arithmetic operation**
  - supply as part of `class` or `struct`

add points

```
struct Point
{
  int x;
  int y;

  public static Point Add(Point p, Point q)
  {
    return new Point(p.x + q.x, p.y + q.y);
  }
  ...
}
```

invoke Add

```
Point a = new Point(1, 2);
Point b = new Point(3, 4);

Point c = Point.Add(a, b);
```

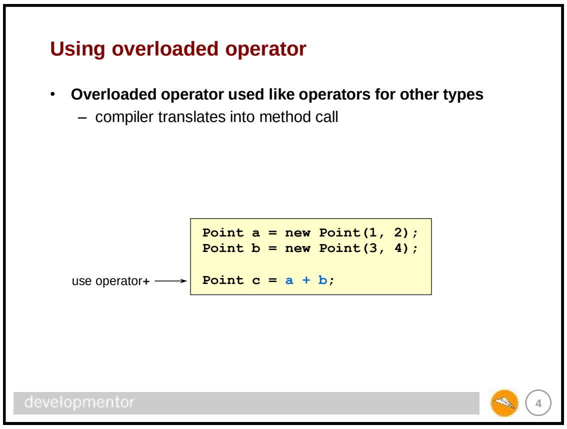


Operator overloading

- **Can overload operators to work with `class` and `struct` types**
  - use keyword `operator`
  - follow with symbol

overload +

```
struct Point
{
  int x;
  int y;

  public static Point operator+(Point p, Point q)
  {
    return new Point(p.x + q.x, p.y + q.y);
  }
  ...
}
```

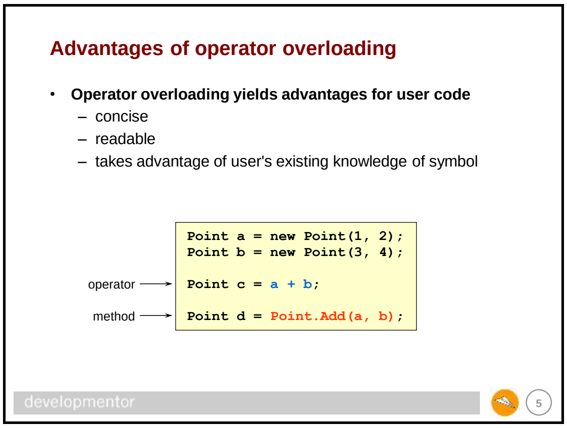


Using overloaded operator

- **Overloaded operator used like operators for other types**
  - compiler translates into method call

use operator+

```
Point a = new Point(1, 2);
Point b = new Point(3, 4);

Point c = a + b;
```



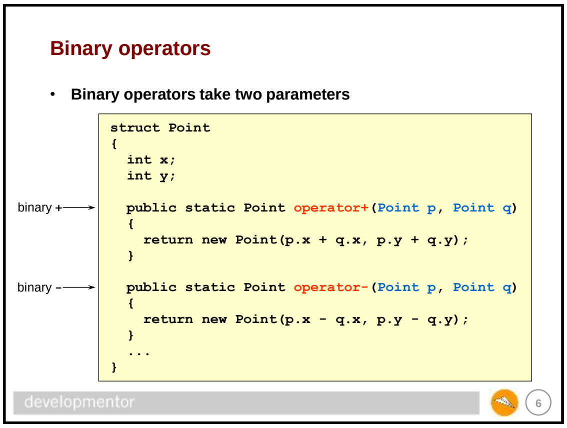
Advantages of operator overloading

- **Operator overloading yields advantages for user code**
  - concise
  - readable
  - takes advantage of user's existing knowledge of symbol

operator

method

```
Point a = new Point(1, 2);
Point b = new Point(3, 4);

Point c = a + b;

Point d = Point.Add(a, b);
```



Binary operators

- **Binary operators take two parameters**

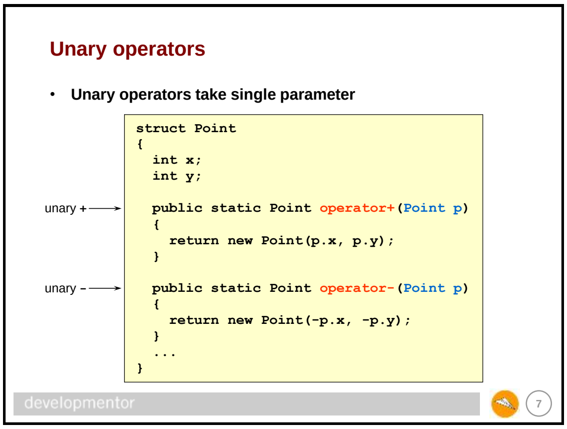
binary +

binary -

```
struct Point
{
  int x;
  int y;

  public static Point operator+(Point p, Point q)
  {
    return new Point(p.x + q.x, p.y + q.y);
  }

  public static Point operator-(Point p, Point q)
  {
    return new Point(p.x - q.x, p.y - q.y);
  }
  ...
}
```

21: Operator Overloading

12/01/2003



Unary operators

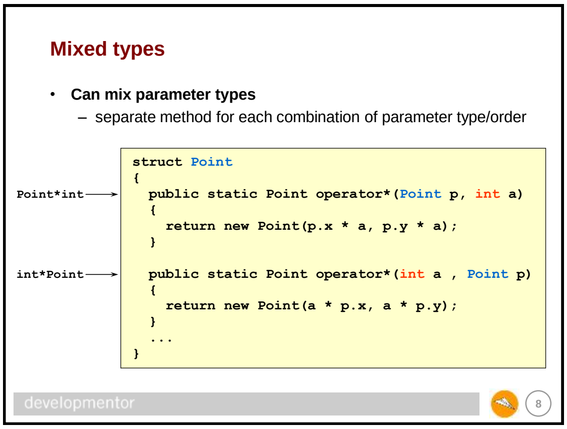
- **Unary operators take single parameter**

unary + →

unary - →

```
struct Point
{
  int x;
  int y;

  public static Point operator+(Point p)
  {
    return new Point(p.x, p.y);
  }

  public static Point operator-(Point p)
  {
    return new Point(-p.x, -p.y);
  }
  ...
}
```

developmentor 7



Mixed types

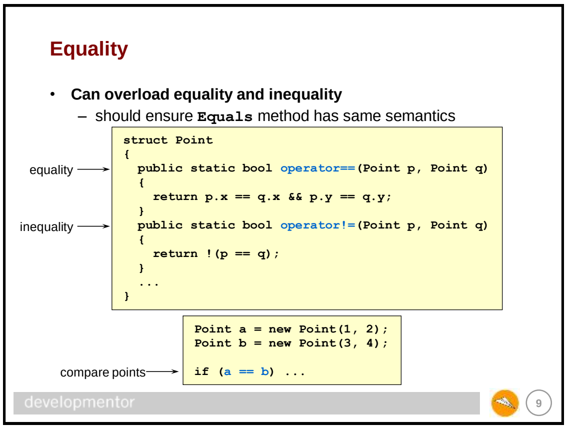
- **Can mix parameter types**
  - separate method for each combination of parameter type/order

Point*int →

int*Point →

```
struct Point
{
  public static Point operator*(Point p, int a)
  {
    return new Point(p.x * a, p.y * a);
  }

  public static Point operator*(int a , Point p)
  {
    return new Point(a * p.x, a * p.y);
  }
  ...
}
```

developmentor 8



Equality

- **Can overload equality and inequality**
  - should ensure `Equals` method has same semantics

equality →

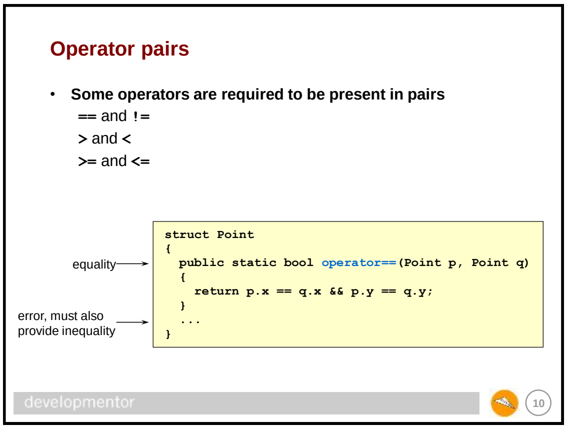
inequality →

```
struct Point
{
  public static bool operator==(Point p, Point q)
  {
    return p.x == q.x && p.y == q.y;
  }
  public static bool operator!=(Point p, Point q)
  {
    return !(p == q);
  }
  ...
}
```

compare points →

```
Point a = new Point(1, 2);
Point b = new Point(3, 4);

if (a == b) ...
```

developmentor 9



Operator pairs

- **Some operators are required to be present in pairs**

  == and !=

  > and <

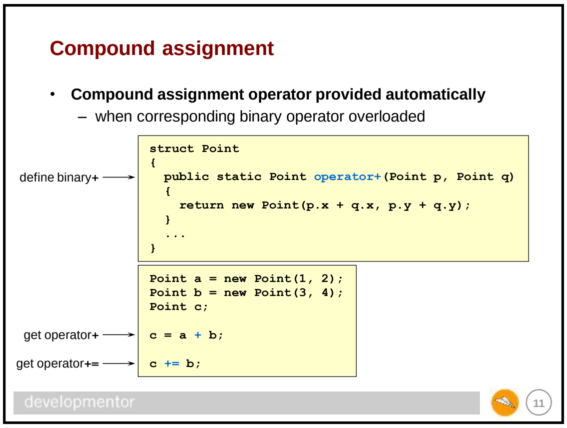
  >= and <=

equality →

error, must also provide inequality →

```
struct Point
{
  public static bool operator==(Point p, Point q)
  {
    return p.x == q.x && p.y == q.y;
  }
  ...
}
```

developmentor 10



Compound assignment

- **Compound assignment operator provided automatically**
  - when corresponding binary operator overloaded

define binary+ →

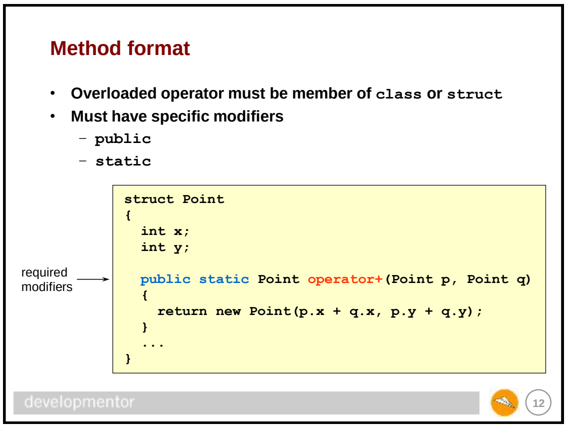
```
struct Point
{
  public static Point operator+(Point p, Point q)
  {
    return new Point(p.x + q.x, p.y + q.y);
  }
  ...
}
```

get operator+ →

get operator+= →

```
Point a = new Point(1, 2);
Point b = new Point(3, 4);
Point c;

c = a + b;

c += b;
```

developmentor 11



Method format

- **Overloaded operator must be member of `class` or `struct`**
- **Must have specific modifiers**
  - `public`
  - `static`

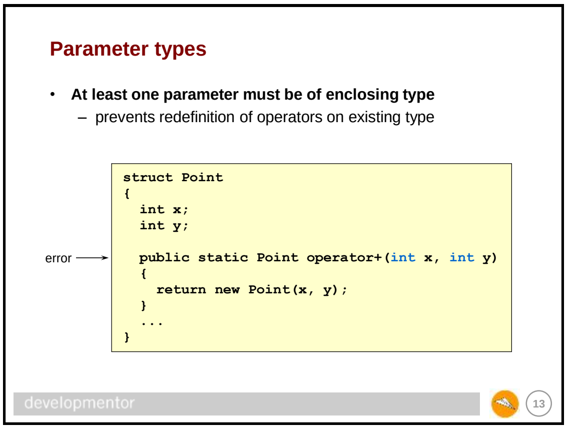
required modifiers →

```
struct Point
{
  int x;
  int y;

  public static Point operator+(Point p, Point q)
  {
    return new Point(p.x + q.x, p.y + q.y);
  }
  ...
}
```

developmentor 12

21: Operator Overloading

12/01/2003

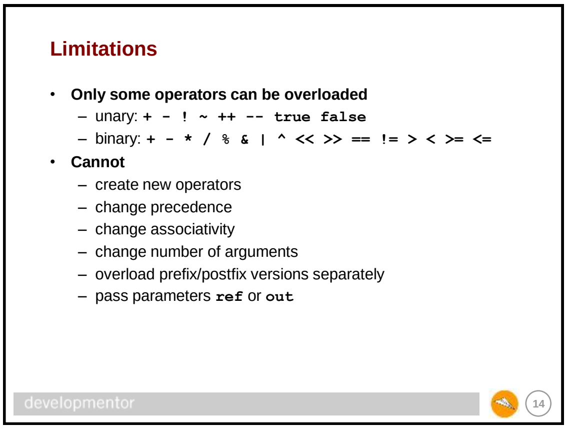


Parameter types

- **At least one parameter must be of enclosing type**
  – prevents redefinition of operators on existing type

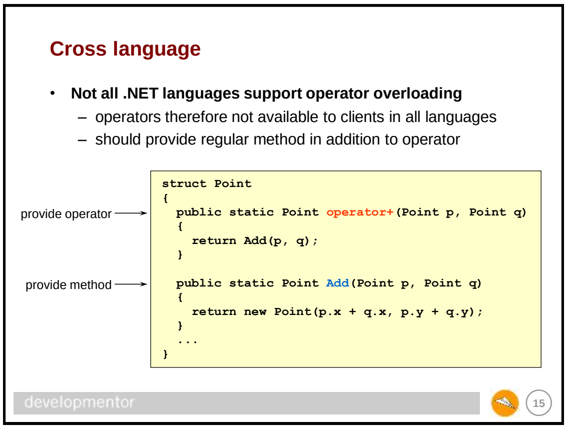
```
struct Point
{
  int x;
  int y;

  public static Point operator+(int x, int y)
  {
    return new Point(x, y);
  }
  ...
}
```

error →



Limitations

- **Only some operators can be overloaded**
  – unary: `+ - ! ~ ++ -- true false`
  – binary: `+ - * / % & | ^ << >> == != > < >= <=`
- **Cannot**
  – create new operators
  – change precedence
  – change associativity
  – change number of arguments
  – overload prefix/postfix versions separately
  – pass parameters `ref` or `out`



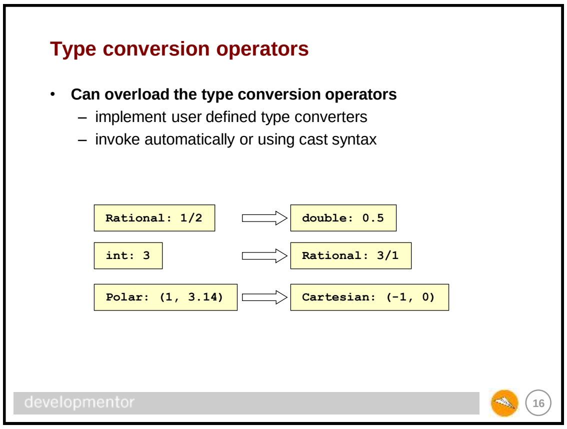
Cross language

- **Not all .NET languages support operator overloading**
  – operators therefore not available to clients in all languages
  – should provide regular method in addition to operator

```
struct Point
{
  public static Point operator+(Point p, Point q)
  {
    return Add(p, q);
  }

  public static Point Add(Point p, Point q)
  {
    return new Point(p.x + q.x, p.y + q.y);
  }
  ...
}
```

provide operator →

provide method →



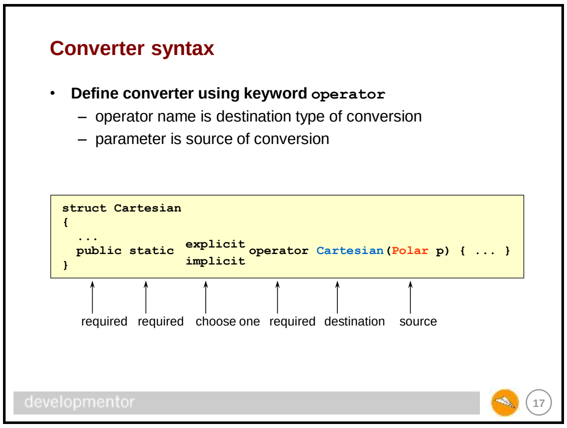
Type conversion operators

- **Can overload the type conversion operators**
  – implement user defined type converters
  – invoke automatically or using cast syntax

`Rational: 1/2` ⇨ `double: 0.5`

`int: 3` ⇨ `Rational: 3/1`

`Polar: (1, 3.14)` ⇨ `Cartesian: (-1, 0)`

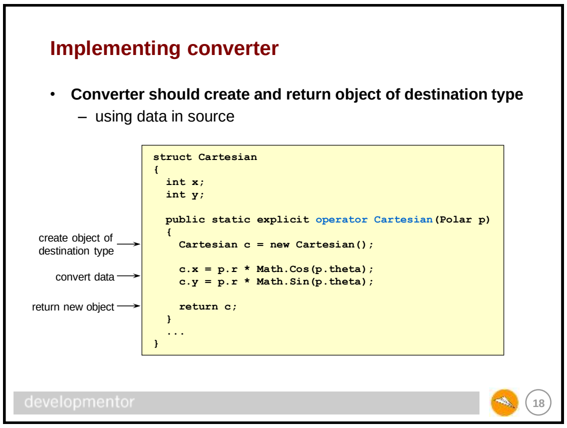


Converter syntax

- **Define converter using keyword `operator`**
  – operator name is destination type of conversion
  – parameter is source of conversion

```
struct Cartesian
{
  ...
                 explicit
  public static           operator Cartesian(Polar p) { ... }
                 implicit
}
```

required · required · choose one · required · destination · source



Implementing converter

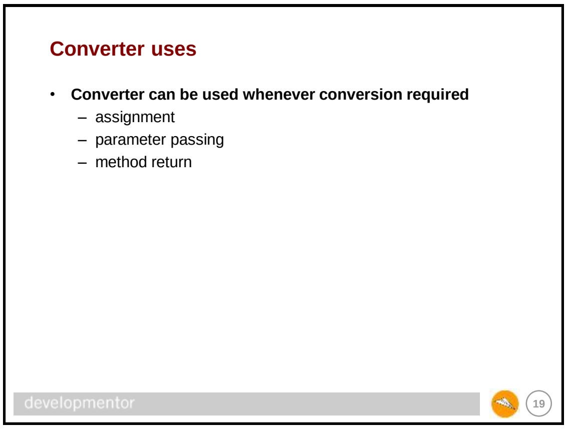
- **Converter should create and return object of destination type**
  – using data in source

```
struct Cartesian
{
  int x;
  int y;

  public static explicit operator Cartesian(Polar p)
  {
    Cartesian c = new Cartesian();

    c.x = p.r * Math.Cos(p.theta);
    c.y = p.r * Math.Sin(p.theta);

    return c;
  }
  ...
}
```

create object of destination type →

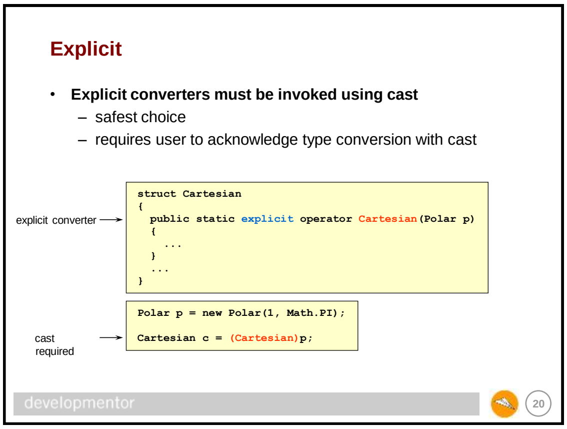
convert data →

return new object →



Converter uses

- **Converter can be used whenever conversion required**
  - assignment
  - parameter passing
  - method return



Explicit

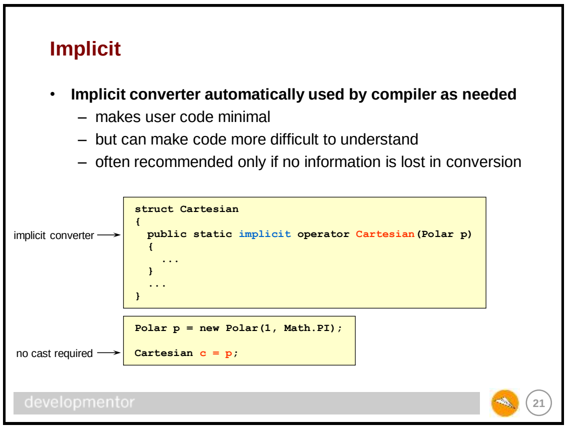
- **Explicit converters must be invoked using cast**
  - safest choice
  - requires user to acknowledge type conversion with cast

explicit converter

```
struct Cartesian
{
  public static explicit operator Cartesian(Polar p)
  {
    ...
  }
  ...
}
```

cast required

```
Polar p = new Polar(1, Math.PI);

Cartesian c = (Cartesian)p;
```



Implicit

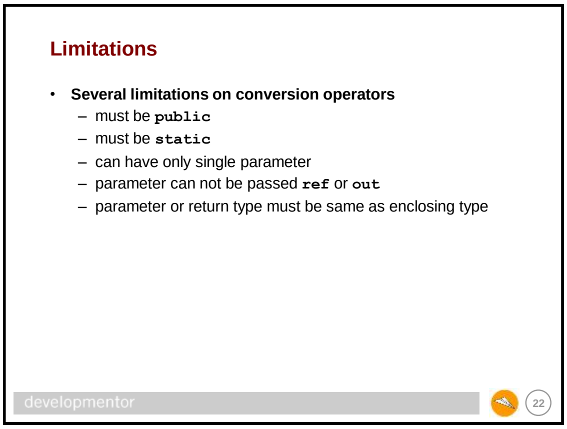
- **Implicit converter automatically used by compiler as needed**
  - makes user code minimal
  - but can make code more difficult to understand
  - often recommended only if no information is lost in conversion

implicit converter

```
struct Cartesian
{
  public static implicit operator Cartesian(Polar p)
  {
    ...
  }
  ...
}
```

no cast required

```
Polar p = new Polar(1, Math.PI);

Cartesian c = p;
```



Limitations

- **Several limitations on conversion operators**
  - must be `public`
  - must be `static`
  - can have only single parameter
  - parameter can not be passed `ref` or `out`
  - parameter or return type must be same as enclosing type