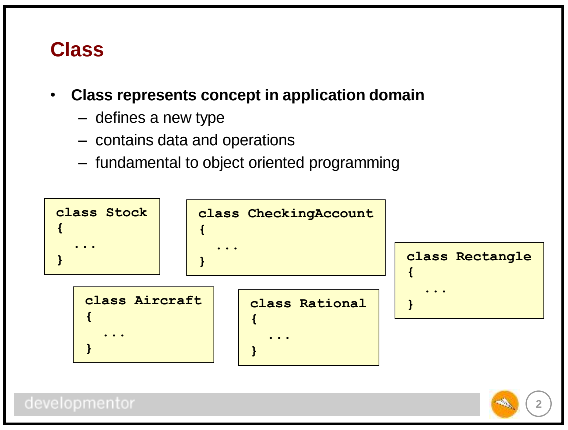
3: Class

12/01/2003

Class

(modified)

developmentor



Class

- **Class represents concept in application domain**
  - defines a new type
  - contains data and operations
  - fundamental to object oriented programming

```
class Stock
{
  ...
}
```

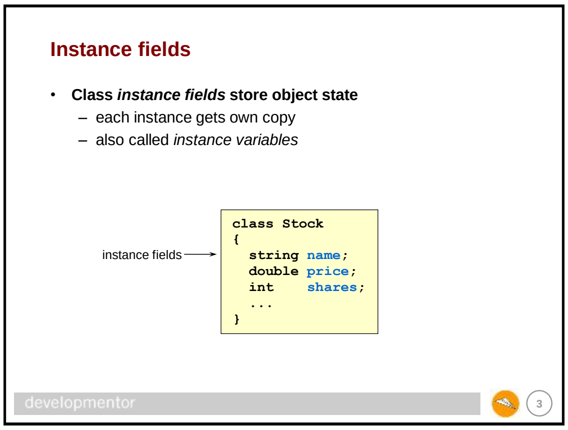
```
class CheckingAccount
{
  ...
}
```

```
class Rectangle
{
  ...
}
```

```
class Aircraft
{
  ...
}
```

```
class Rational
{
  ...
}
```

developmentor 2



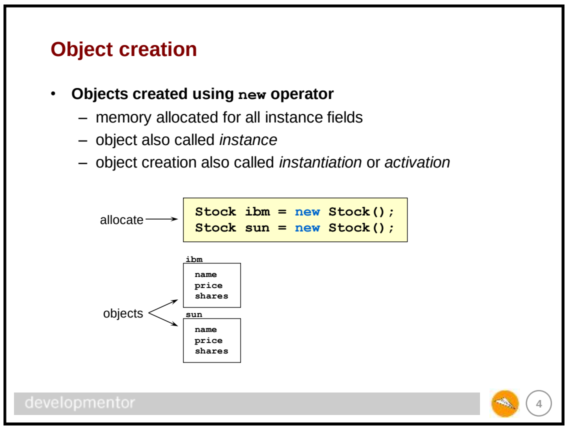
Instance fields

- **Class *instance fields* store object state**
  - each instance gets own copy
  - also called *instance variables*

instance fields →

```
class Stock
{
  string name;
  double price;
  int    shares;
  ...
}
```

developmentor 3



Object creation

- **Objects created using `new` operator**
  - memory allocated for all instance fields
  - object also called *instance*
  - object creation also called *instantiation* or *activation*

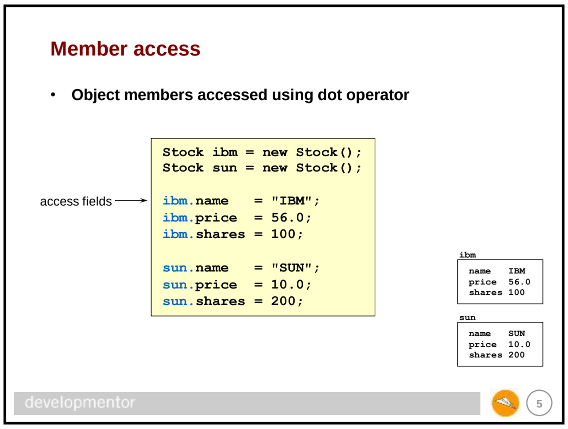
allocate →

```
Stock ibm = new Stock();
Stock sun = new Stock();
```

objects

ibm
name
price
shares
sun
name
price
shares

developmentor 4



Member access

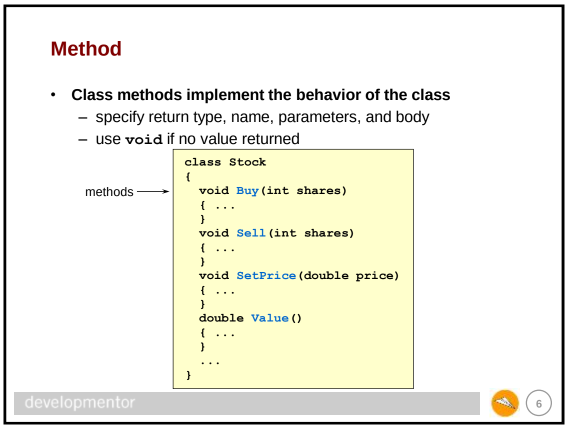
- **Object members accessed using dot operator**

access fields →

```
Stock ibm = new Stock();
Stock sun = new Stock();

ibm.name   = "IBM";
ibm.price  = 56.0;
ibm.shares = 100;

sun.name   = "SUN";
sun.price  = 10.0;
sun.shares = 200;
```

ibm
name IBM
price 56.0
shares 100
sun
name SUN
price 10.0
shares 200

developmentor 5



Method

- **Class methods implement the behavior of the class**
  - specify return type, name, parameters, and body
  - use `void` if no value returned

methods →

```
class Stock
{
  void Buy(int shares)
  { ...
  }
  void Sell(int shares)
  { ...
  }
  void SetPrice(double price)
  { ...
  }
  double Value()
  { ...
  }
  ...
}
```

developmentor 6

Method invocation

- Method invoked on object
  - use dot operator
  - pass parameters

call methods →

```
Stock ibm = new Stock();

ibm.SetPrice(56.0);
ibm.Buy(100);

ibm.SetPrice(99.0);
ibm.Sell(50);
```

developmentor 7

Method implementation

- Method implementation uses keyword `this`
  - handle to object on which method was called
  - used to access members from inside method

implement method →

use `this` →

```
class Stock
{
  string name;
  double price;
  int    shares;

  void Buy(int s)
  {
    this.shares += s;
  }
  ...
}
```

```
Stock ibm = new Stock();

ibm.SetPrice(56.0);
ibm.Buy(100);
```

```
ibm/this
 name
 price   56.0
 shares  100
```

developmentor 8

Required `this`

- Must use `this` when local name conflicts with member name

parameter and field have same name →

must use `this` →

```
class Stock
{
  string name;
  double price;
  int    shares;

  void Buy(int shares)
  {
    this.shares += shares;
  }
  ...
}
```

developmentor 9

Omitting this

- Can omit `this` when there is no ambiguity
  - class members accessed implicitly

omit `this` →

```
class Stock
{
  string name;
  double price;
  int    shares;

  void Buy(int s)
  {
    shares += s;
  }
  ...
}
```

developmentor 10

Member declaration order

- Members may be declared in any order
  - no requirement to declare before use
  - different rule than local variables

use →

declare →

```
class Stock
{
  void Buy(int s)
  {
    shares += s;
  }

  string name;
  double price;
  int    shares;
  ...
}
```

developmentor 11

Return value

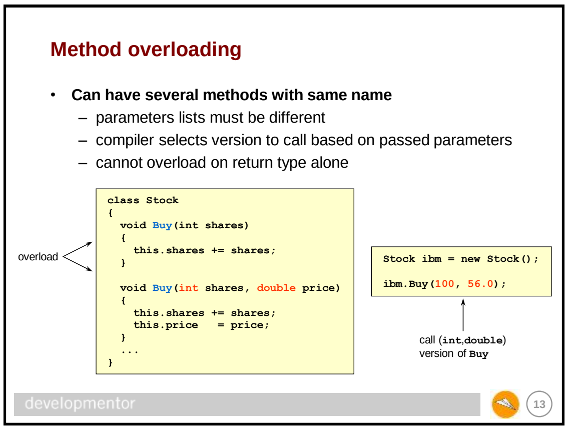
- Use `return` to send data out of method
  - value sent back to caller

return →

```
class Stock
{
  double Value()
  {
    return price * shares;
  }
  ...
}
```

```
Stock ibm = new Stock();

ibm.SetPrice(56.0);
ibm.Buy(100);

double v = ibm.Value();
```

developmentor 12



Method overloading

- **Can have several methods with same name**
  - parameters lists must be different
  - compiler selects version to call based on passed parameters
  - cannot overload on return type alone

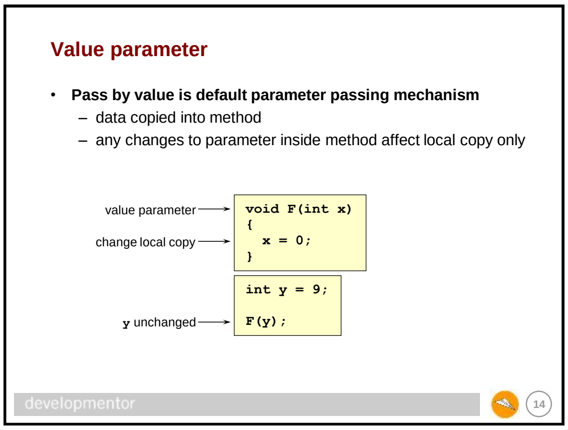
overload

```
class Stock
{
  void Buy(int shares)
  {
    this.shares += shares;
  }

  void Buy(int shares, double price)
  {
    this.shares += shares;
    this.price  = price;
  }
  ...
}
```

```
Stock ibm = new Stock();

ibm.Buy(100, 56.0);
```

call (`int`,`double`) version of `Buy`

developmentor 13



Value parameter

- **Pass by value is default parameter passing mechanism**
  - data copied into method
  - any changes to parameter inside method affect local copy only

value parameter

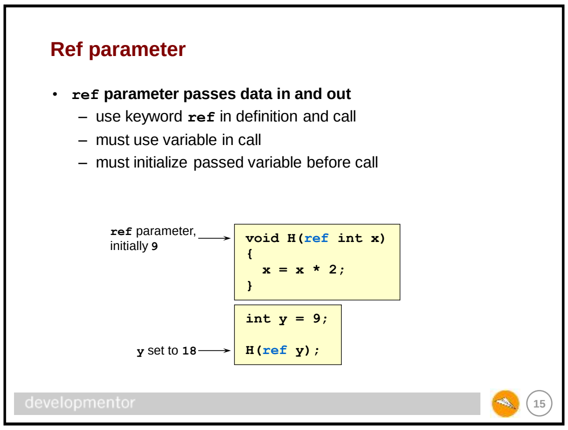
change local copy

```
void F(int x)
{
  x = 0;
}
```

`y` unchanged

```
int y = 9;

F(y);
```

developmentor 14



Ref parameter

- **`ref` parameter passes data in and out**
  - use keyword `ref` in definition and call
  - must use variable in call
  - must initialize passed variable before call

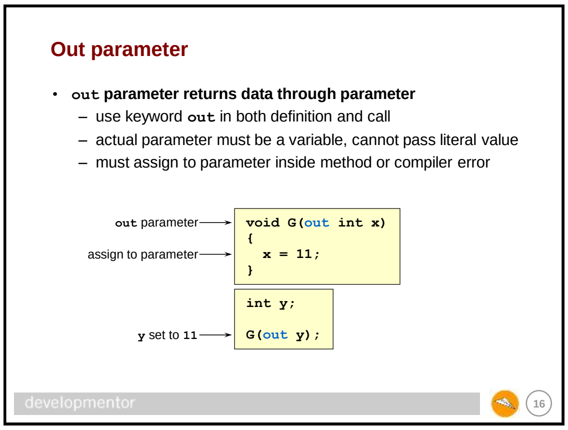
`ref` parameter, initially `9`

```
void H(ref int x)
{
  x = x * 2;
}
```

`y` set to `18`

```
int y = 9;

H(ref y);
```

developmentor 15



Out parameter

- **`out` parameter returns data through parameter**
  - use keyword `out` in both definition and call
  - actual parameter must be a variable, cannot pass literal value
  - must assign to parameter inside method or compiler error

`out` parameter

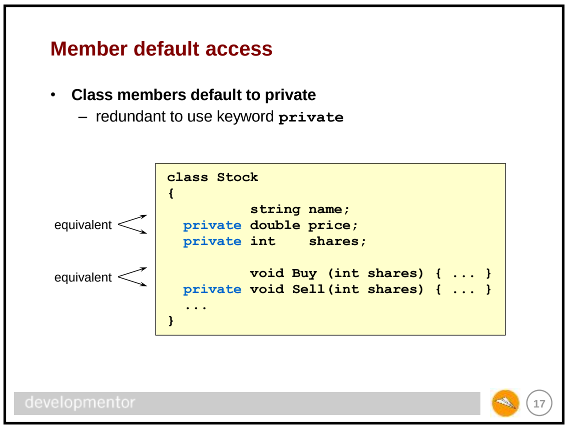
assign to parameter

```
void G(out int x)
{
  x = 11;
}
```

`y` set to `11`

```
int y;

G(out y);
```

developmentor 16



Member default access

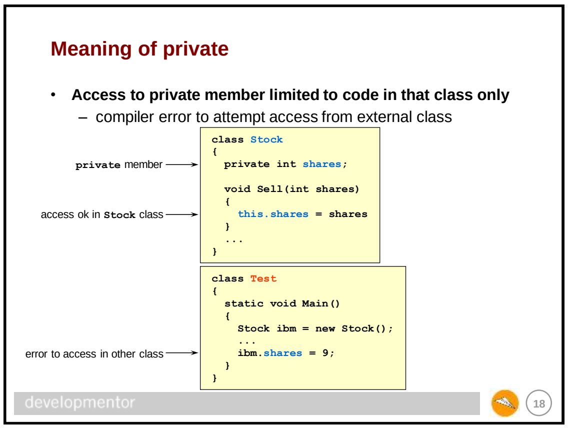
- **Class members default to private**
  - redundant to use keyword `private`

equivalent

equivalent

```
class Stock
{
          string name;
  private double price;
  private int    shares;

          void Buy (int shares) { ... }
  private void Sell(int shares) { ... }
  ...
}
```

developmentor 17



Meaning of private

- **Access to private member limited to code in that class only**
  - compiler error to attempt access from external class

`private` member

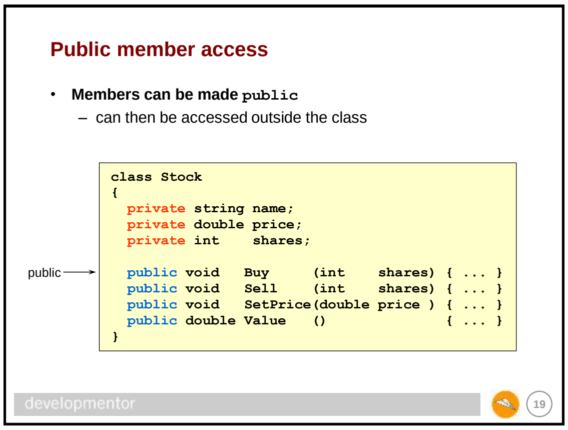
access ok in `Stock` class

```
class Stock
{
  private int shares;

  void Sell(int shares)
  {
    this.shares = shares
  }
  ...
}
```

error to access in other class

```
class Test
{
  static void Main()
  {
    Stock ibm = new Stock();
    ...
    ibm.shares = 9;
  }
}
```

developmentor 18

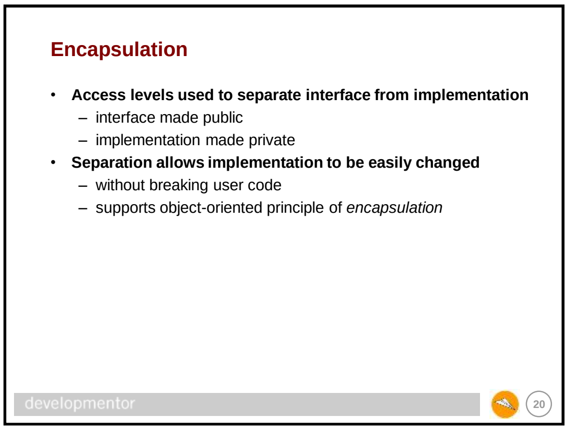


Public member access

- **Members can be made `public`**
  - can then be accessed outside the class

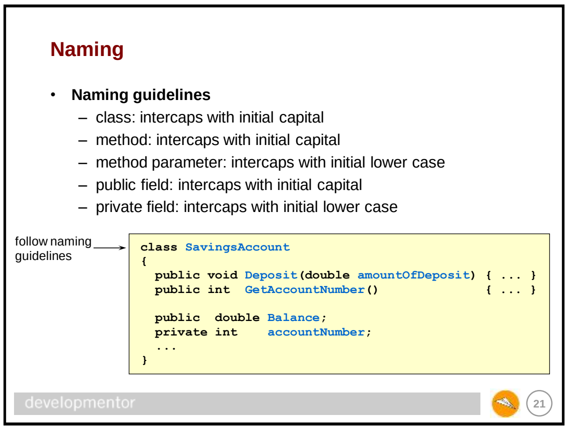
public →

```
class Stock
{
  private string name;
  private double price;
  private int    shares;

  public void   Buy      (int    shares) { ... }
  public void   Sell     (int    shares) { ... }
  public void   SetPrice(double price ) { ... }
  public double Value    ()              { ... }
}
```



Encapsulation

- **Access levels used to separate interface from implementation**
  - interface made public
  - implementation made private
- **Separation allows implementation to be easily changed**
  - without breaking user code
  - supports object-oriented principle of *encapsulation*



Naming

- **Naming guidelines**
  - class: intercaps with initial capital
  - method: intercaps with initial capital
  - method parameter: intercaps with initial lower case
  - public field: intercaps with initial capital
  - private field: intercaps with initial lower case

follow naming guidelines →

```
class SavingsAccount
{
  public void Deposit(double amountOfDeposit) { ... }
  public int  GetAccountNumber()              { ... }

  public  double Balance;
  private int    accountNumber;
  ...
}
```