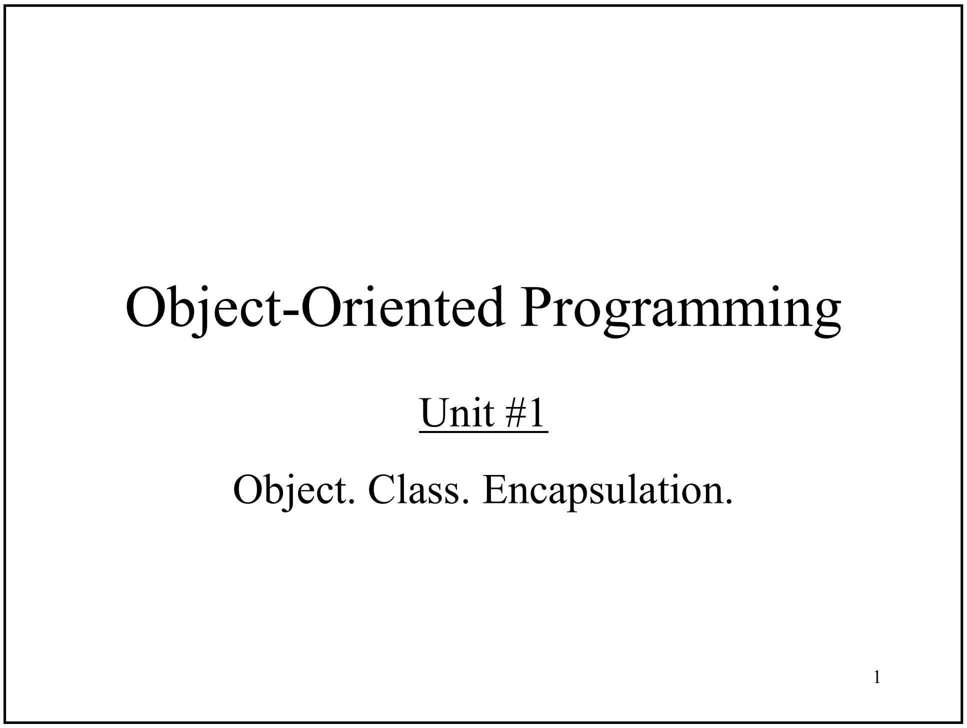
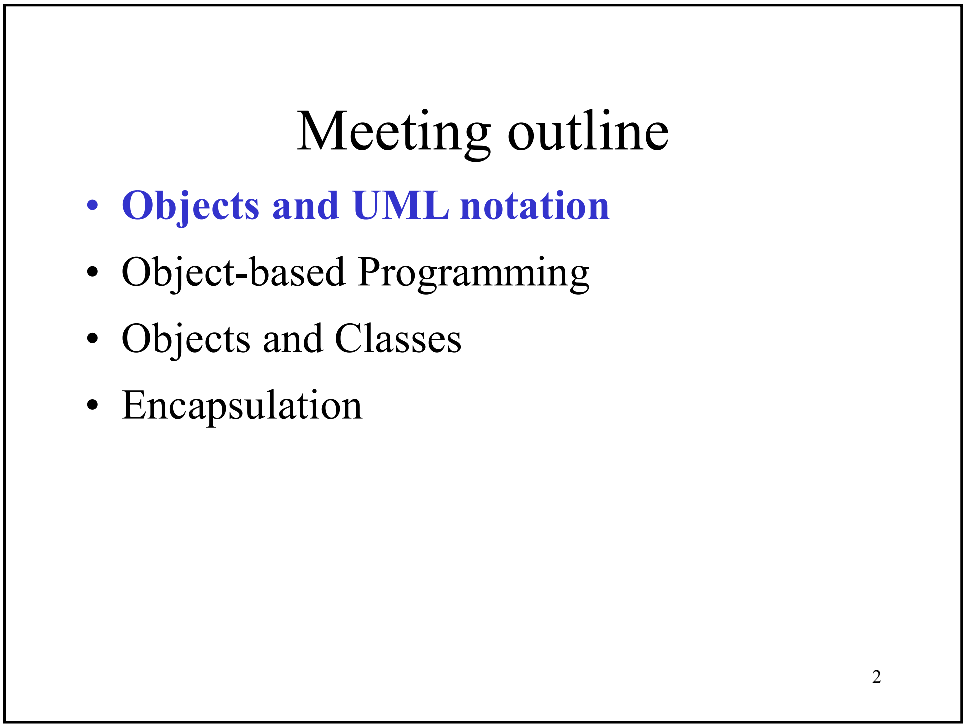


Object-Oriented Programming

Unit #1

Object. Class. Encapsulation.



Meeting outline

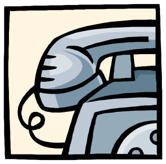
- **Objects and UML notation**
- Object-based Programming
- Objects and Classes
- Encapsulation

Learning Outcomes

- Learning the very basic OO definitions
- Understanding the encapsulation principle
- Getting familiar with UML notation
- Building your own classes and objects with UML notation

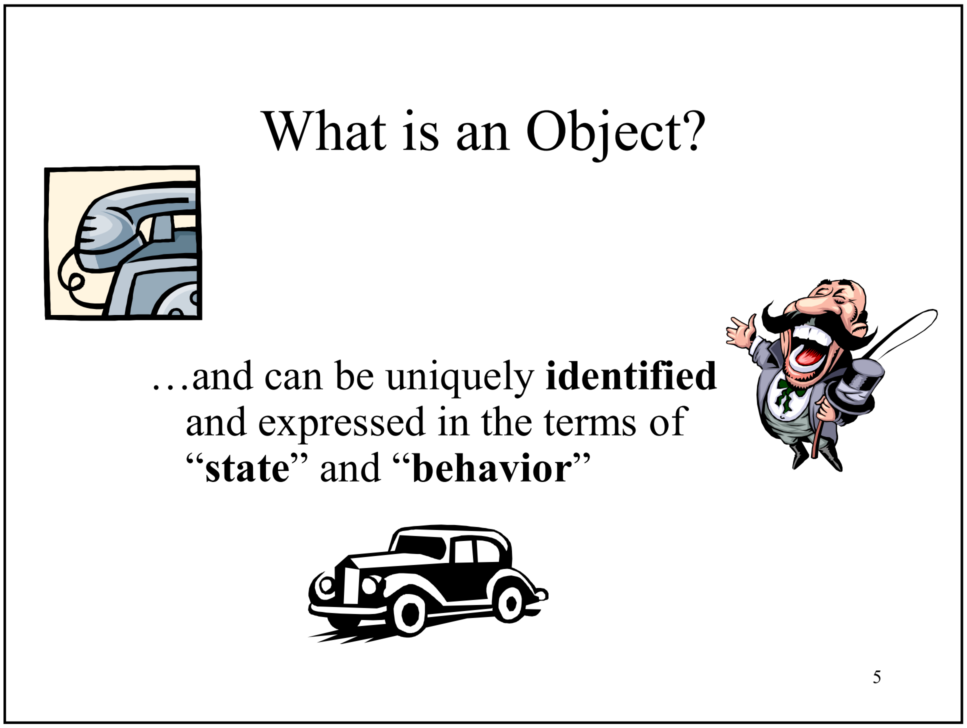
3

What is an Object?



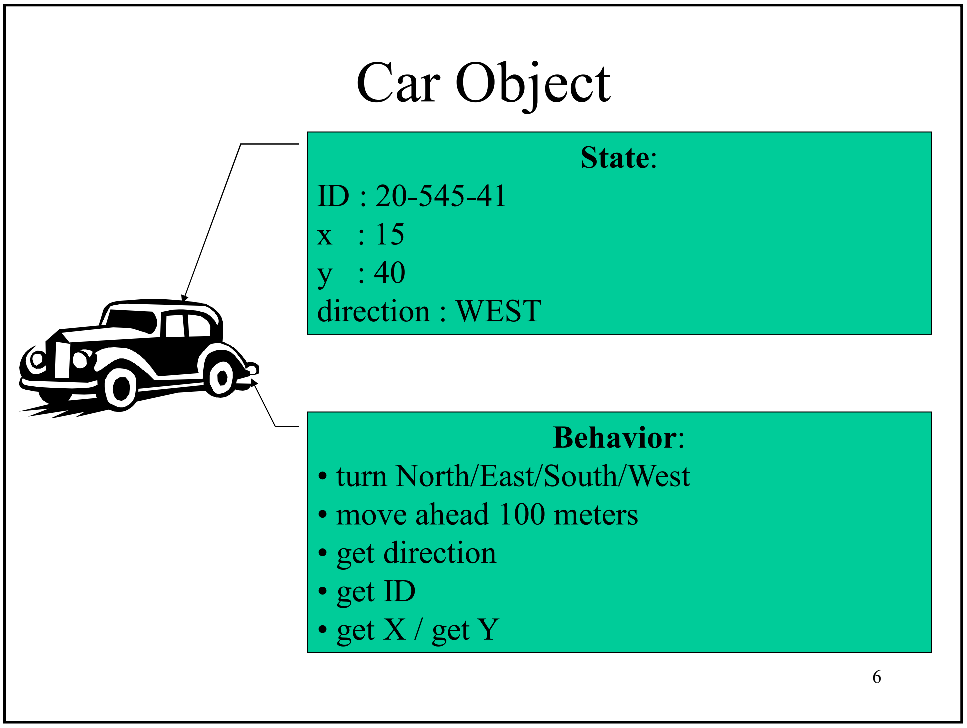
Software object represents a domain-dependent copy of a real world object…

4



What is an Object?

…and can be uniquely **identified** and expressed in the terms of “**state**” and “**behavior**”



Car Object

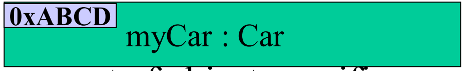
State:
ID : 20-545-41
x : 15
y : 40
direction : WEST

Behavior:

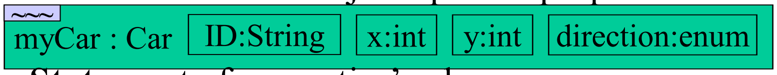
- turn North/East/South/West
- move ahead 100 meters
- get direction
- get ID
- get X / get Y

Basic Definitions

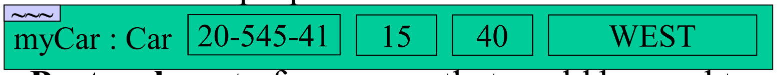
- **Identity** – a unique object identifier



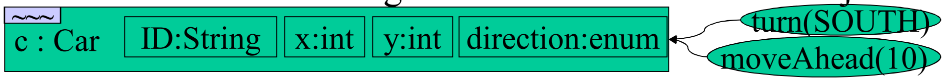
- **Structure** – set of object-specific properties



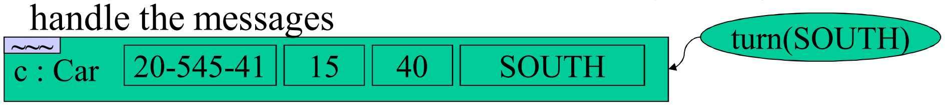
- **State** – set of properties’ values



- **Protocol** – set of messages that could be send to an object



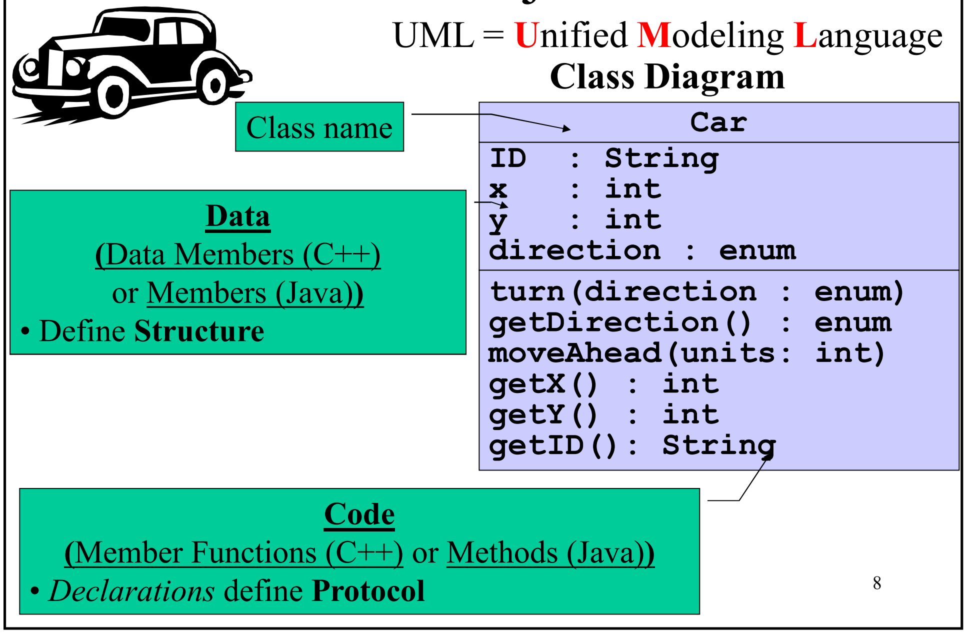
- **Behavior** – set of operations performed by an object to handle the messages



Car Object

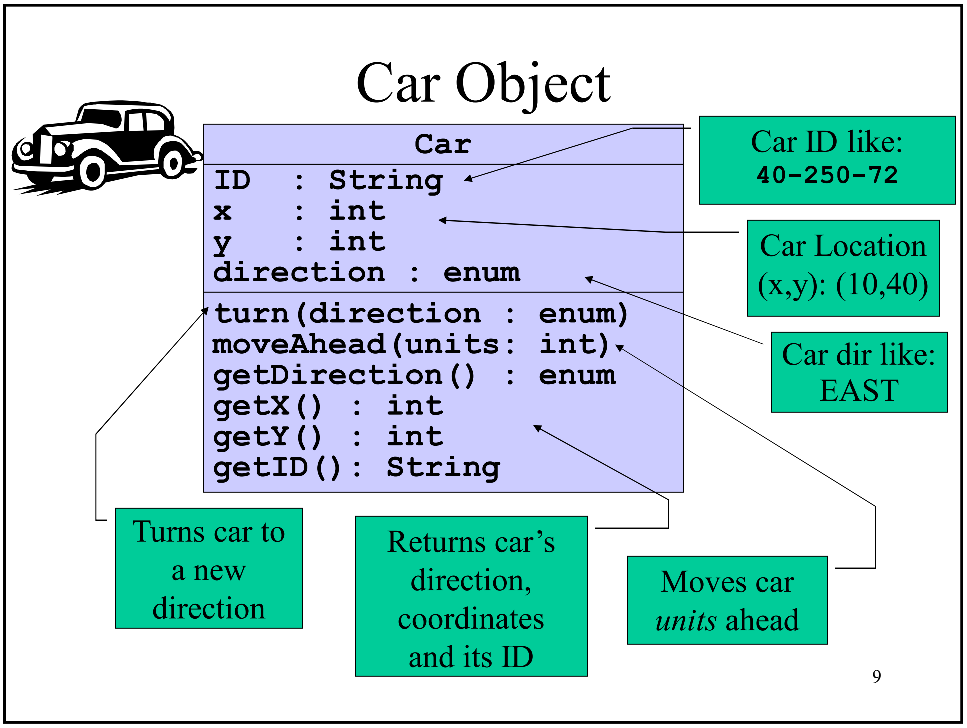
UML = Unified Modeling Language

Class Diagram



Data
(Data Members (C++)
or Members (Java))
- Define **Structure**

Code
(Member Functions (C++) or Methods (Java))
- *Declarations* define **Protocol**



Car Object

```
Car
ID   : String
x    : int
y    : int
direction : enum
turn(direction : enum)
moveAhead(units: int)
getDirection() : enum
getX() : int
getY() : int
getID(): String
```

Car ID like: **40-250-72**

Car Location (x,y): (10,40)

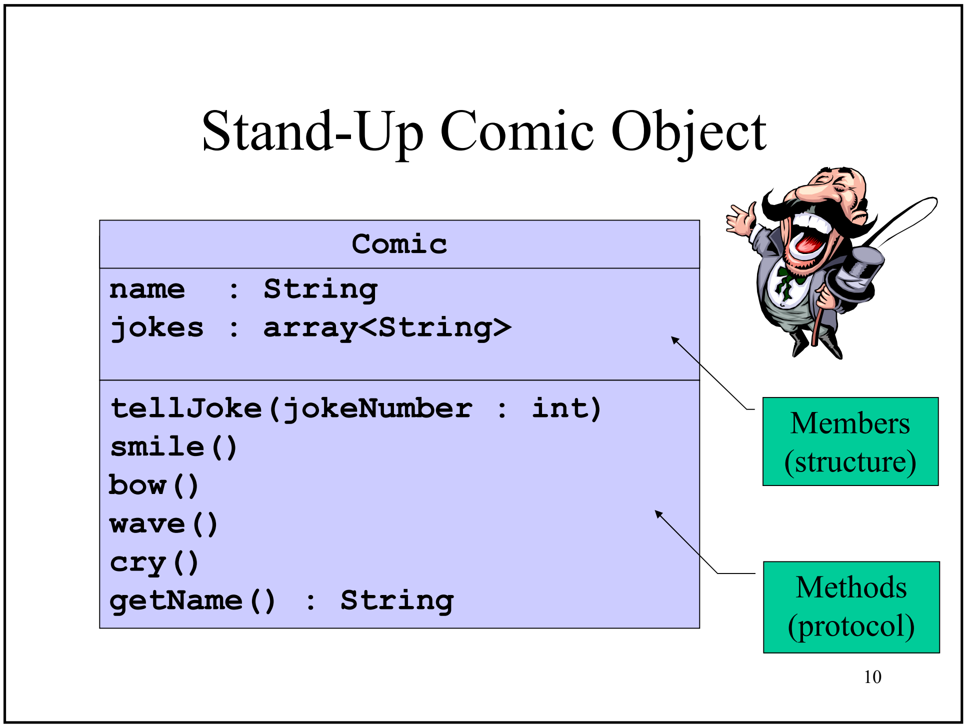
Car dir like: EAST

Turns car to a new direction

Returns car's direction, coordinates and its ID

Moves car *units* ahead

9



Stand-Up Comic Object

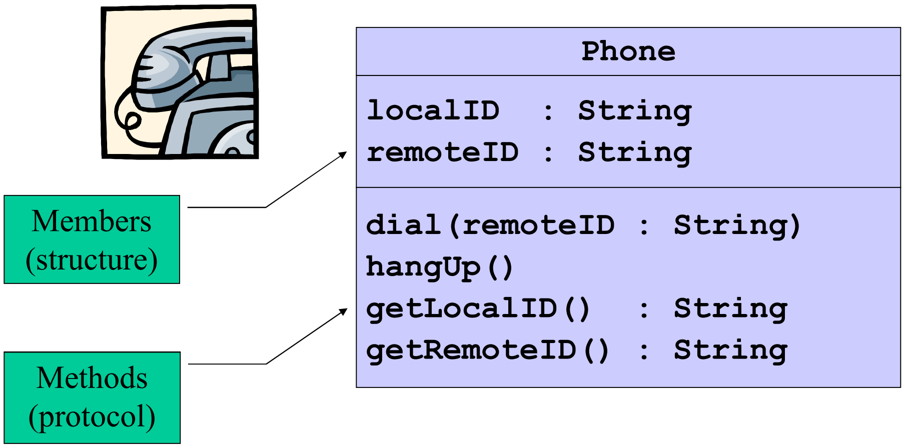
```
Comic
name  : String
jokes : array<String>

tellJoke(jokeNumber : int)
smile()
bow()
wave()
cry()
getName() : String
```

Members (structure)

Methods (protocol)

10

Phone Object



```
Phone
localID   : String
remoteID  : String
dial(remoteID : String)
hangUp()
getLocalID()  : String
getRemoteID() : String
```

Members (structure)

Methods (protocol)

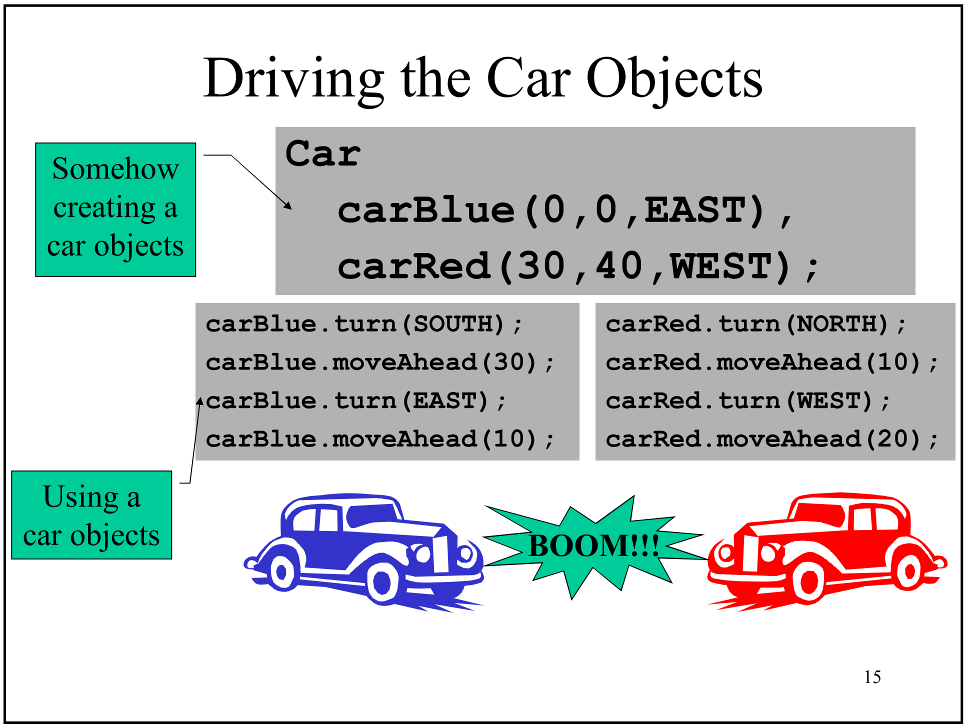
Objects are Everywhere

Meeting outline

- Objects and UML notation
- **Object-based Programming**
- Objects and Classes
- Encapsulation

Driving a car in real life

- Get car (at given location)
- Drive
  - Change direction
  - Move ahead
- Stop



Driving the Car Objects

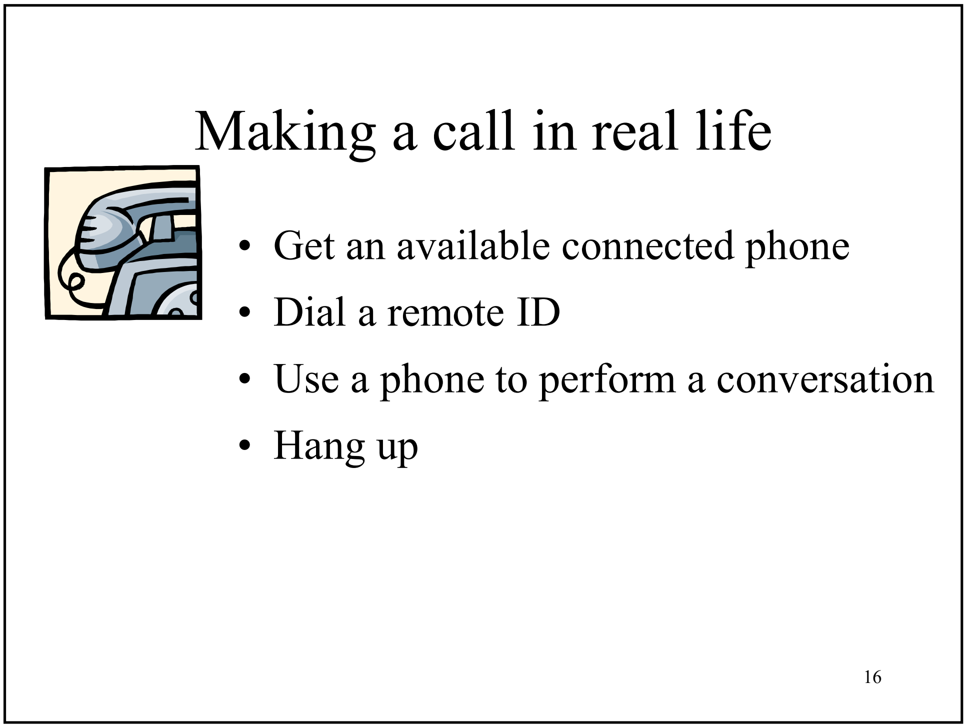
Somehow creating a car objects

```
Car
  carBlue(0,0,EAST),
  carRed(30,40,WEST);
```

```
carBlue.turn(SOUTH);
carBlue.moveAhead(30);
carBlue.turn(EAST);
carBlue.moveAhead(10);
```

```
carRed.turn(NORTH);
carRed.moveAhead(10);
carRed.turn(WEST);
carRed.moveAhead(20);
```

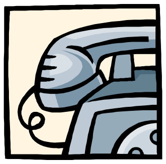
Using a car objects



Making a call in real life

- Get an available connected phone
- Dial a remote ID
- Use a phone to perform a conversation
- Hang up

Making a call with a Phone Object



Phone's object **identity**

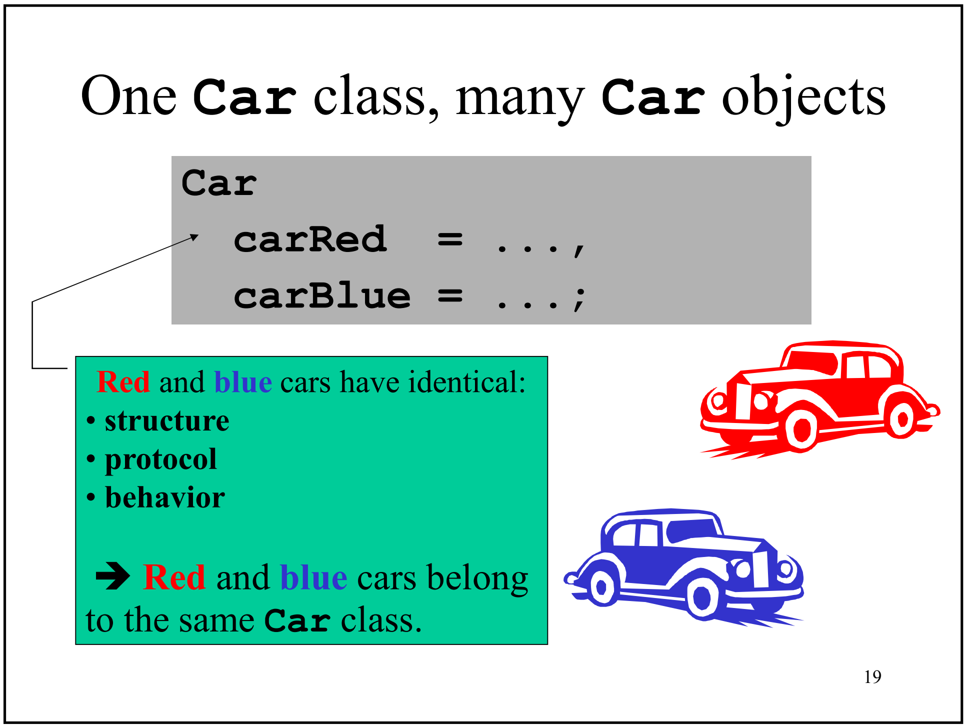
```
Phone phoneAtHome =
  Phone(“972-3-6948888”);
phoneAtHome.dial(“972-3-7521133”);

…make a conversation… (shhhh!)

phoneAtHome.hangUp();
```

Creating a phone object

Using a phone object:
calling **code** to manipulate the **data**

Meeting outline

- Objects and UML notation
- Object-based Programming
- **Objects and Classes**
- Encapsulation

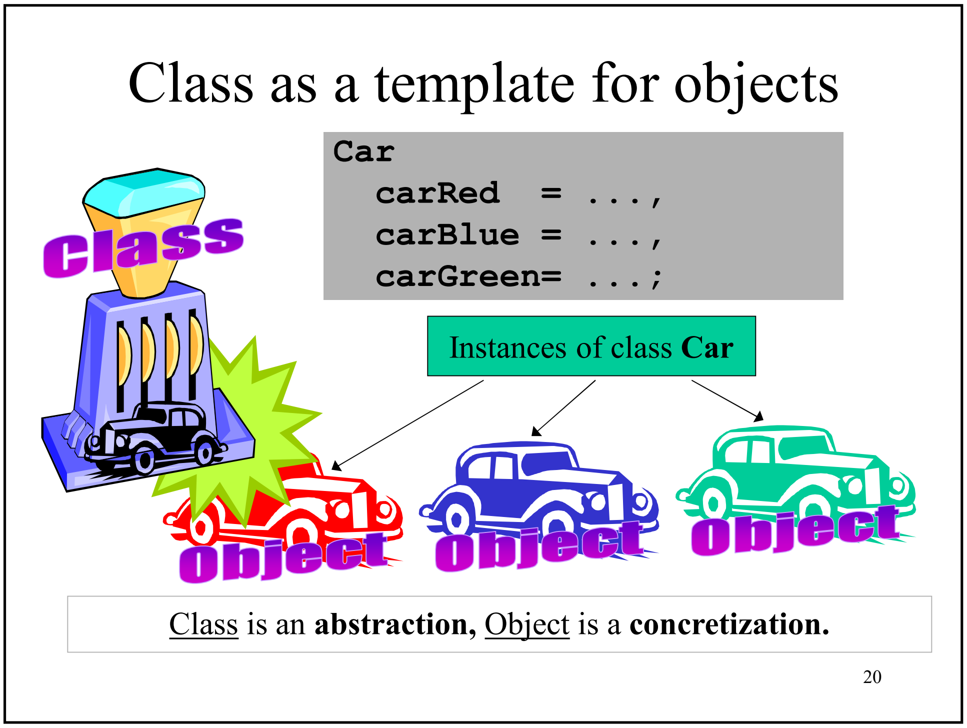


One **Car** class, many **Car** objects

```
Car
   carRed  = ...,
   carBlue = ...;
```

Red and blue cars have identical:
- **structure**
- **protocol**
- **behavior**

➔ Red and blue cars belong to the same **Car** class.



Class as a template for objects

```
Car
   carRed   = ...,
   carBlue  = ...,
   carGreen= ...;
```

Instances of class **Car**

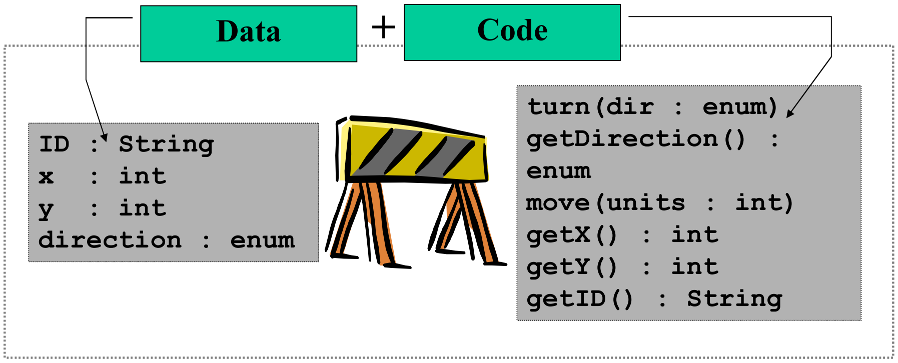
Class is an **abstraction,** Object is a **concretization.**

Meeting Outline

- Objects and UML notation
- Object-based Programming
- Object and Class
- **Encapsulation**

Encapsulation
(Implementation details hiding)

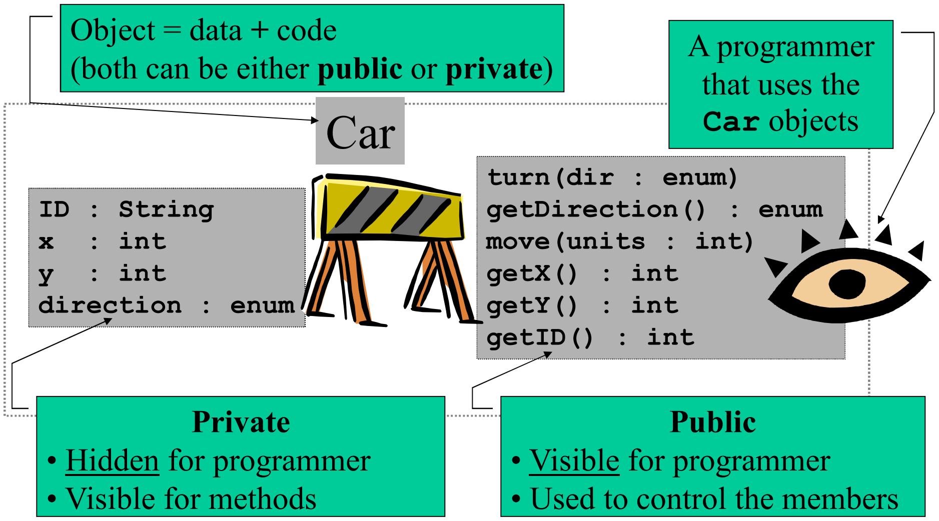
Car Object =



```
ID : String
x  : int
y  : int
direction : enum
```

```
turn(dir : enum)
getDirection() :
enum
move(units : int)
getX() : int
getY() : int
getID() : String
```

Encapsulation
(Implementation details hiding) - cont



Object = data + code
(both can be either **public** or **private**)

A programmer that uses the **Car** objects

Car

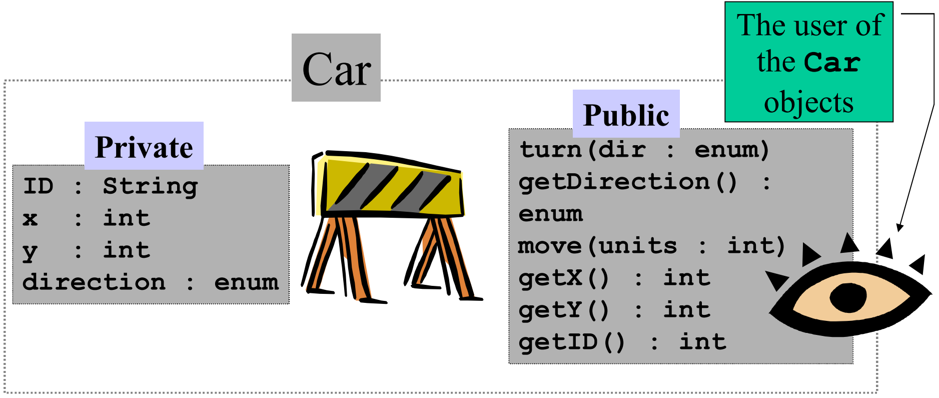
```
ID : String
x  : int
y  : int
direction : enum
```

```
turn(dir : enum)
getDirection() : enum
move(units : int)
getX() : int
getY() : int
getID() : int
```

Private
- Hidden for programmer
- Visible for methods

Public
- Visible for programmer
- Used to control the members

Encapsulation (cont)



Car

The user of the **Car** objects

Private

```
ID : String
x  : int
y  : int
direction : enum
```

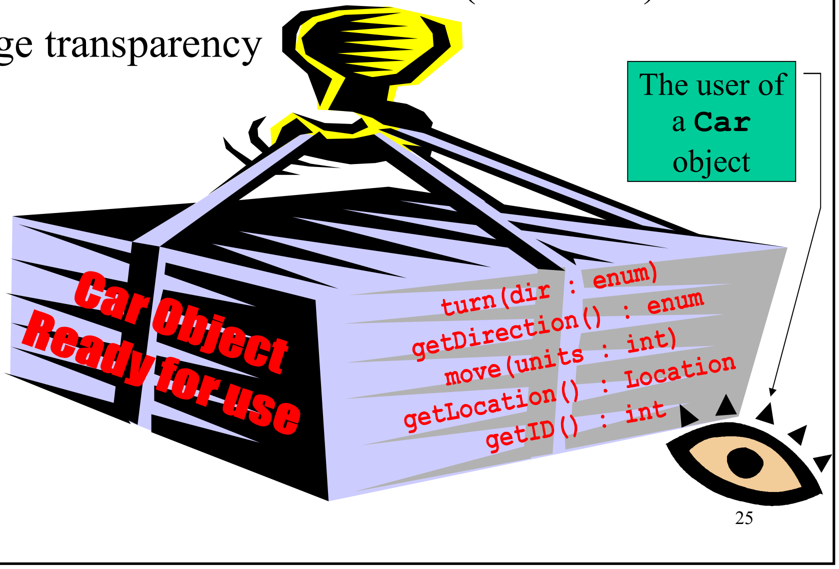
Public

```
turn(dir : enum)
getDirection() :
enum
move(units : int)
getX() : int
getY() : int
getID() : int
```

- **Minimum exposure**: the user of the object knows nothing about the implementation details.
- **Maximum functionality**: the user of the objects receives all the necessary service.

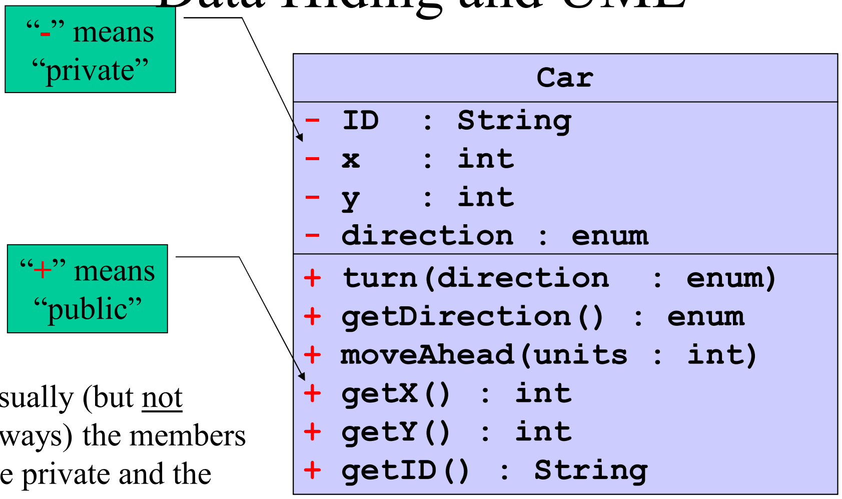
Benefits of encapsulation

- Object as a self-consistent unit (black box)
- Usage transparency



The user of a **Car** object

Data Hiding and UML

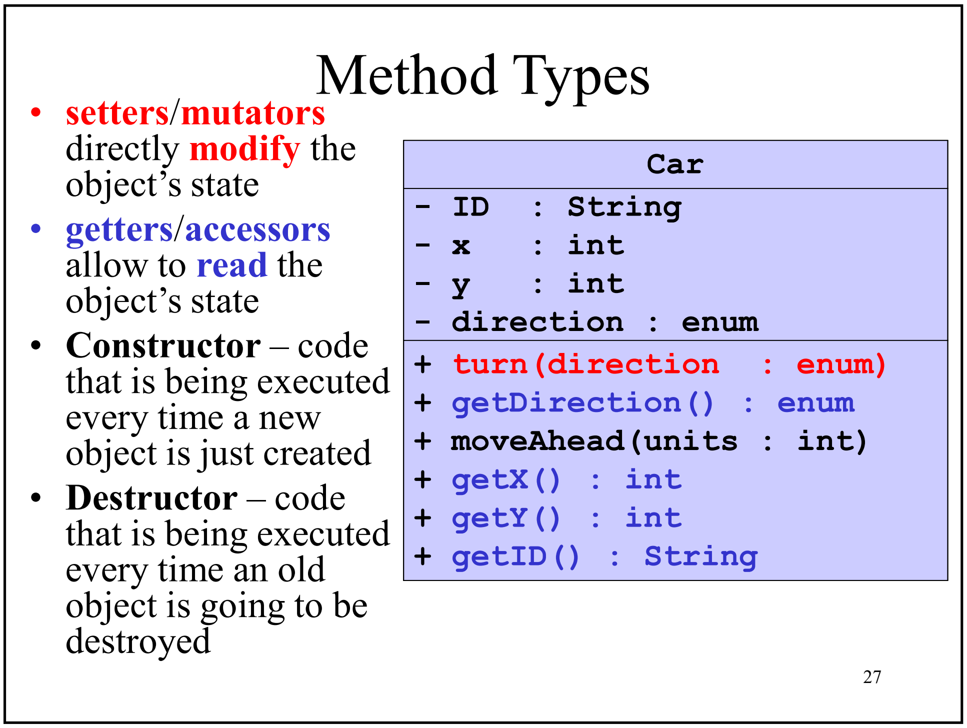


"-" means "private"

"+" means "public"

```
                    Car
- ID    : String
- x     : int
- y     : int
- direction : enum
+ turn(direction   : enum)
+ getDirection() : enum
+ moveAhead(units : int)
+ getX() : int
+ getY() : int
+ getID() : String
```

Usually (but <u>not</u> always) the members are private and the methods are public

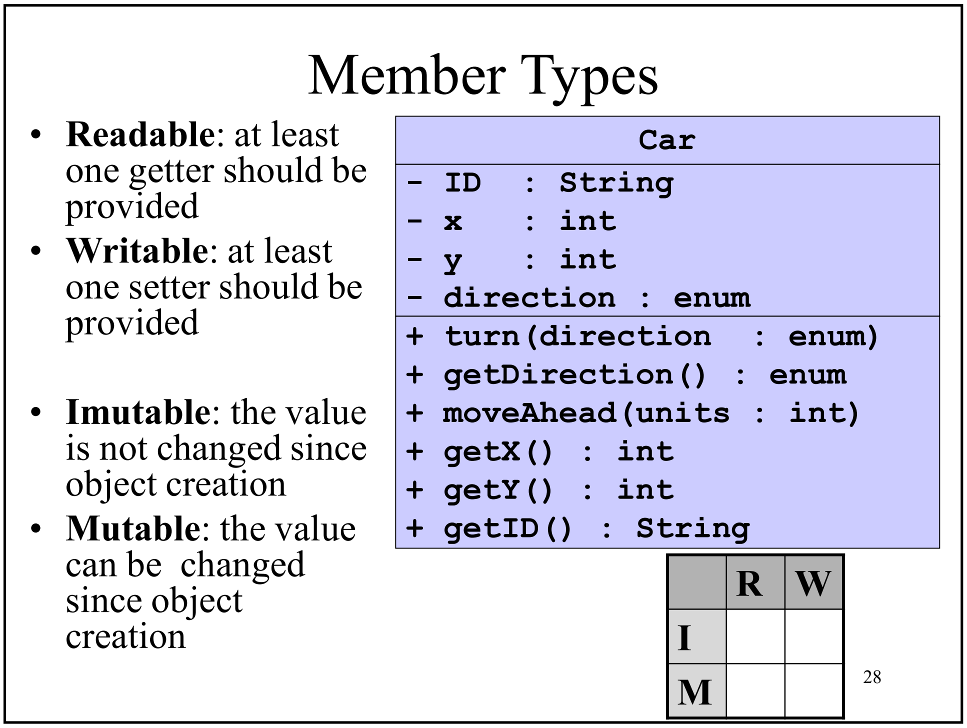


Method Types

- **setters/mutators** directly **modify** the object’s state
- **getters/accessors** allow to **read** the object’s state
- **Constructor** – code that is being executed every time a new object is just created
- **Destructor** – code that is being executed every time an old object is going to be destroyed

```
                Car
- ID   : String
- x    : int
- y    : int
- direction : enum
+ turn(direction  : enum)
+ getDirection() : enum
+ moveAhead(units : int)
+ getX() : int
+ getY() : int
+ getID() : String
```

27



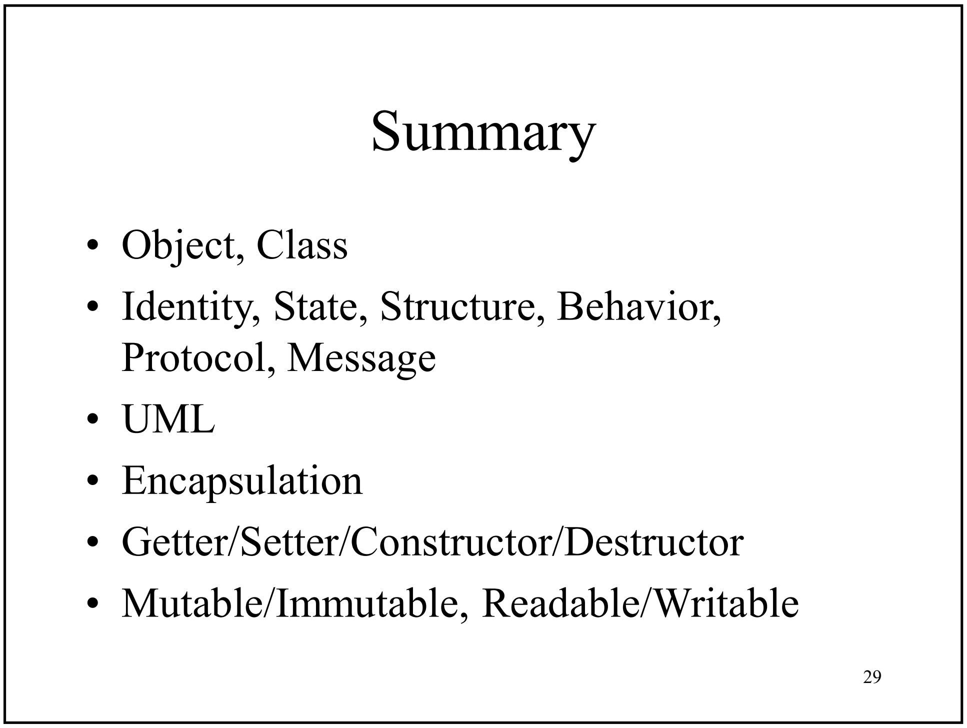
Member Types

- **Readable**: at least one getter should be provided
- **Writable**: at least one setter should be provided
- **Imutable**: the value is not changed since object creation
- **Mutable**: the value can be changed since object creation

```
                Car
- ID   : String
- x    : int
- y    : int
- direction : enum
+ turn(direction  : enum)
+ getDirection() : enum
+ moveAhead(units : int)
+ getX() : int
+ getY() : int
+ getID() : String
```

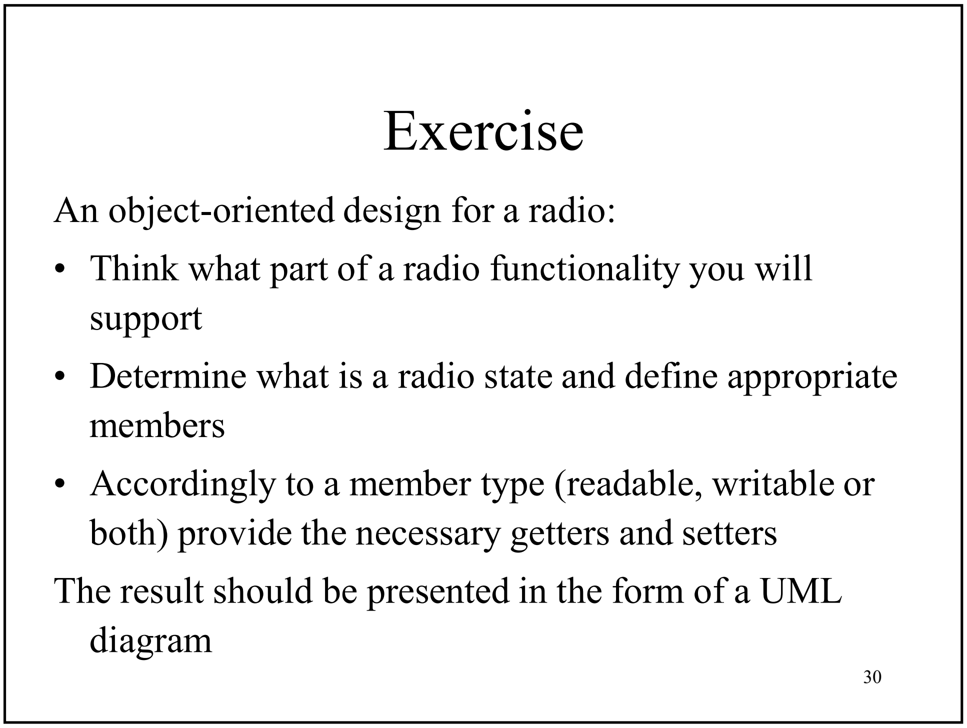
	R	W
I		
M		

28



Summary

- Object, Class
- Identity, State, Structure, Behavior, Protocol, Message
- UML
- Encapsulation
- Getter/Setter/Constructor/Destructor
- Mutable/Immutable, Readable/Writable



Exercise

An object-oriented design for a radio:

- Think what part of a radio functionality you will support
- Determine what is a radio state and define appropriate members
- Accordingly to a member type (readable, writable or both) provide the necessary getters and setters

The result should be presented in the form of a UML diagram