# Object Oriented Analysis and Design

# Analysis and Design

- Analysis – **What** does the system need to do? What are the requirements?
- Design – **How** Should the system do it?

# Object Oriented Programming

The three basic elements of OOP are:

- Encapsulation
- Inheritance
- Polymorphism

# Object Oriented Programming (OOP) – Encapsulation

- Encapsulation means dividing the application into several entities.
- Each entity has a well defined role.
- Each entity encapsulates all data and functionality regarding its role within it.

# Object Oriented Programming – Inheritance

- Creating a new entity which is an extension of an existing one.
- Inheritance reflects an ***"is a"*** relationship.
- A derived (inherited) class is actually its ancestor plus additional data and/or functionality.
- Inheritance also enables us to change simultaneously the basic structure of several different entities.

# Object Oriented Programming – Polymorphism

- Polymorphism is the ability of a reference to act differently on different occasions.
- References may be assigned to objects of either the reference class or to objects which are descendants of that class.
- A reference of a certain class might change its behavior according to the object type it references.

# Classes as Blueprints for Objects

- In manufacturing, a blueprint is a description of a device from which many physical devices are constructed.
- In software, a class is a description of an object.
- The class describes both the data (data members) of an object and its behavior (the methods it holds).
- Objects are instances of a class.

# Declaring Java Classes

- Basic syntax of a java class:
- <class_declaration>::=

```
<modifier> class <name> {
<attribute_declaration>*
<constructor_declaration>*
<method_declaration>*
}
```

- Example:

```
    public class Ship
{
    private String captainName;
     public void setCaptainName (String name) {
             captainName=name;}
}
```

# Declaring Attributes

- <attribute_declaration>::=
  <modifier><type><name> [=<default_value>];
  <type>::= byte | short | int |long | char | float | double | boolean | <class>

- Example:

```
public class Car
{
    private float velocity;
    private float fuelConsumptionPerKm = 11.5;
    private String manufacturer = “Porsche”;
}
```

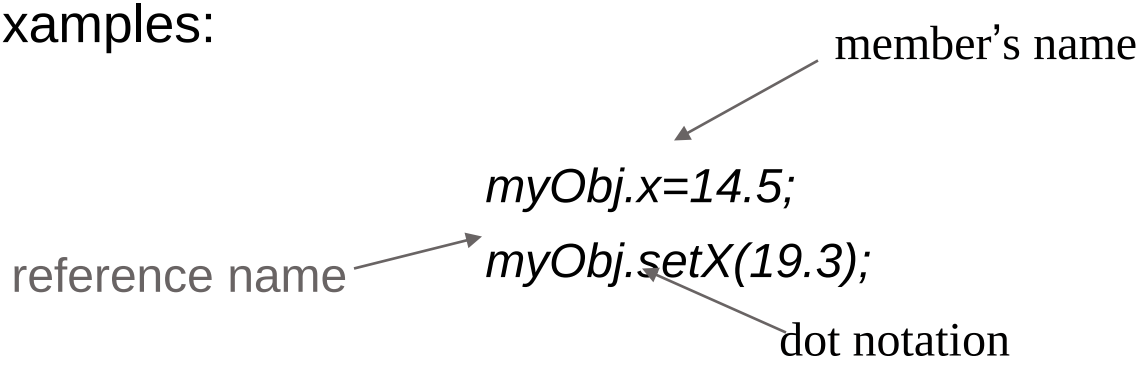
# Declaring Methods

- <method_declaration>::=
  [< modifiers>] <return_type> <name> ([< parameter>]) {
  [< statements>] }
- <parameter>::=
  <parameter type> <parameter_name>
- Example:

```
public class Dog
{
    private int weight;
    public int getWeight() {
          return weight; }
    public void setWeight(int newWeight) {
          weight = newWeight; }
}
```

# Access to Object Members

- Every object member is accessed using the object reference.
- The access is done through the object reference name, the dot notation and the member name.
- Static members are accessed using the class name instead of the reference name.
- Examples:



member's name

*myObj.x=14.5;*

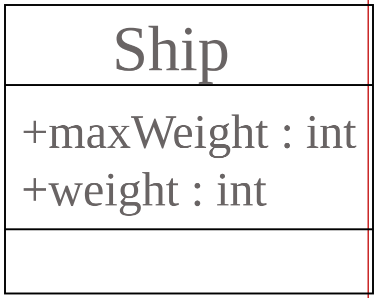
reference name

*myObj.setX(19.3);*

dot notation

- *Note: access is allowed according to the member's modifier.*

# Information Hiding - The Problem

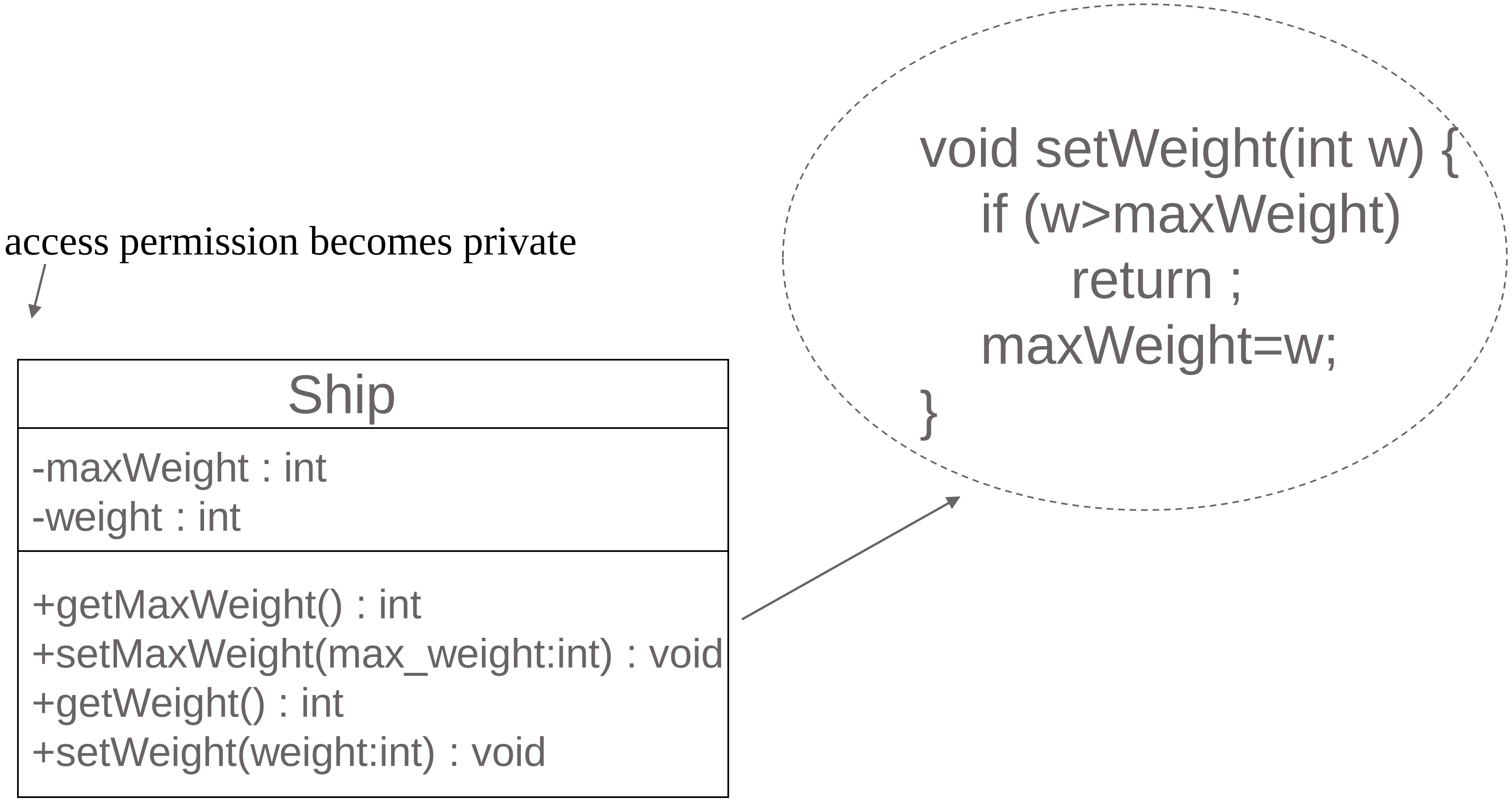


Ship

+maxWeight : int
+weight : int

```
Ship s;
s.maxWeight =340;                          //Max allowed weight is 200.
s. weight =s.maxWeight +100;      //No checking is done.
```

# Information Hiding - The Solution:



access permission becomes private

Ship

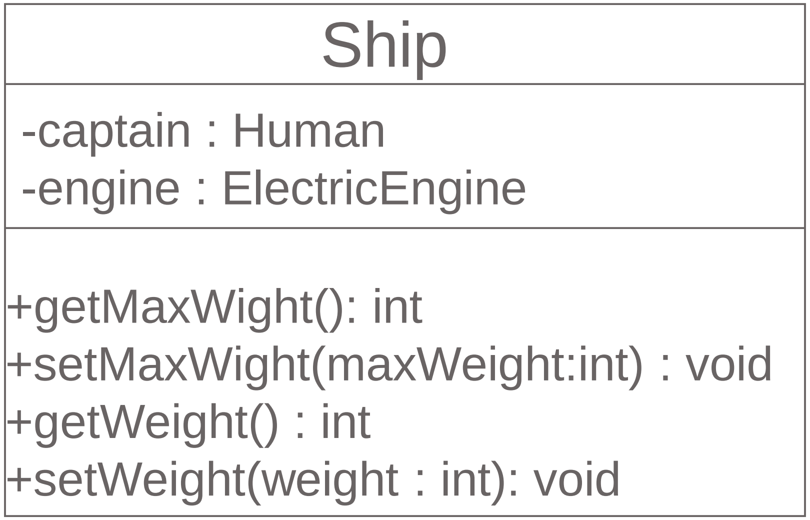
-maxWeight : int
-weight : int

+getMaxWeight() : int
+setMaxWeight(max_weight:int) : void
+getWeight() : int
+setWeight(weight:int) : void

```
void setWeight(int w) {
    if (w>maxWeight)
        return ;
    maxWeight=w;
}
```

# Encapsulation

- Encapsulation of related data and functionality within one class (improve maintainability).
- Hides the implementation details of the class.
- Forces the client to use interfaces for accessing data.



Ship

-captain : Human
-engine : ElectricEngine

+getMaxWight(): int
+setMaxWight(maxWeight:int) : void
+getWeight() : int
+setWeight(weight : int): void

# Constructors

- A method that called right after the creation of an object.
- The constructor usually initializes the object data members.
- Parameters may be sent to a constructor during object creation.

# Constructor Declaration

- <constructor_declaration>::=

```
[< modifier>] <class_name> ([< parameter>]) {
        [< statements>]
  }
```

- Example:

```
public class Cat {
     private int num_of_miyhu;
     public Cat(int m) {
         num_of_miyhu= m; }
     public Cat( ) {
        num_of_miyhu= 3;       //Default value.
    }
    public static void main(String args[]) {
        Cat c1=new Cat(7);
        Cat c2=new Cat( ); }
}
```

# Default Constructor

- Every class has a constructor .
- In case the programmer did not write any constructor, the default constructor will be supplied automatically.
- After the programmer will add a constructor, the default constructor will vanish.
- The default constructor takes no arguments and has no body. It only exists so the call for a constructor made during objects creation will be supplied.

# Packages

- Java classes may be organized into packages.
- Every such package should contain classes that are all related to the same subject (utilities, mathematical functionality, sailing, cargo, etc…).
- The java programming language enables access privileges restricted to the package classes.

# Packages – cont’d

- The default package (no name package) will be used in cases that no package statement appears in the file.
- Packages are stored in the directory tree containing the package name.

# Defining a Package

- <package_declaration>::=

  package <top_package_name>[.<sub_package_name];
- Example:

  *package building.construction.house ;*

- Package declaration should appear only once, at the beginning of a file.
- Packages are hierarchical and are separated by dots.

# Compiling into package

- javac –d <root-location> <sources>

```
javac –d mainDir *.java
```

- Archiving classes / packages into jars:

  jar –cf <result-jar-file-name> <sources>

```
jar –cf  application.jar  mainDir
```

# Import Declarations

- In java, no pre-processor is used. Accordingly there are no header files.
- The *import* statement specifies a path for the compiler to find code, but the class code itself will not be loaded into the file (unlike the C/C++ *#include* statement).
- Import declarations precede all class declarations.
- Basic syntax:

<import_declaration> ::=

    import <pkg_name>[.<sub_pkg_name>]*.<class_name | *>;

- Examples:

import shapes.rectangles.*; //define a path to all classes //in that package.

**import java.lang.*;** //imported always by default

import java.util.List; //define a path to the *List* class.

# Source File Layout

- Basic syntax of a java source file:
- <source_declaration>::=

```
              [<package_declaration>]
              <import_declaration>*
              <class_declaration>+
```

- Example, the *Box.java* file:

```
    package shapes.rectangles;
    import java.util.Map;
    import java.io.*;
    public class Box {
    // Class definition goes here.
  }
```

# Summary

- Class – A blueprint source code for instantiating objects.
- Object – An instance of a class.
- Attribute (Data Member, Instance Variable, Data Field) – A data element of an object.
- Method (Class Function) – A behavioral element of a class.
- Constructor – A method that is called whenever an object of a specific class is instantiated. Used to initialize the data members.
- Package – A grouping of classes (library)