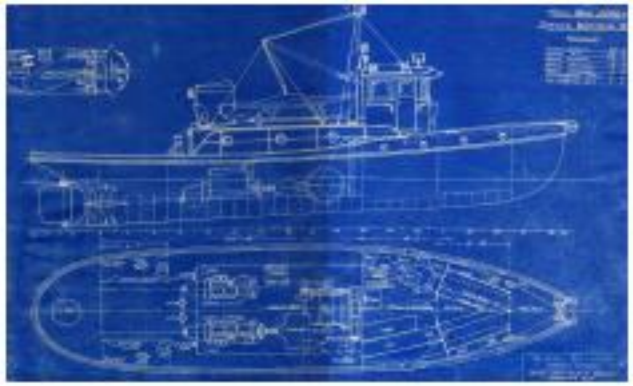
Classes & objects

Agenda

Class

Blueprint of objects



Classes

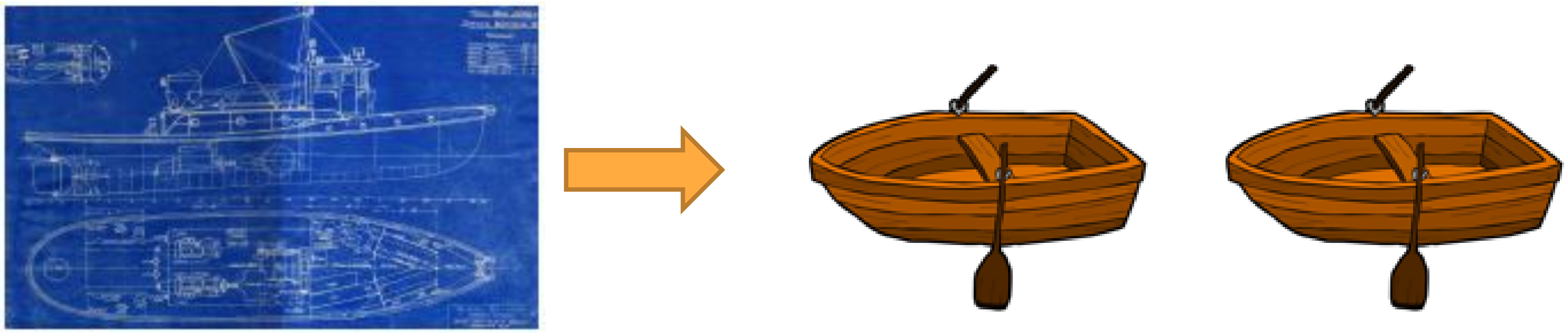
```
[modifiers] class MyClassName {
    // fields,
    // constructors
    // method declarations
}
```

Classes: Example

```
public class Shirt {
  public int shirtID = 0; // Default ID for the shirt
  public Shirt(){
       display();
  } // end of Constructor method
  public void display() {
       System.out.println("Item ID: " + shirtID);
  } // end of display method
} // end of class
```

Objects

Instantiated versions of their class



Objects: Example

Declare and initialize reference

```
Shirt myShirt = new Shirt();
int shirtId = myShirt.shirtId;
myShirt.display();
```

Get the value of `shirtId` field

Call the `display()` method of the object

Classes: Constructors

Constructor

method, invoked upon object creation

Classes: Constructors Example

```
public class Point {
    private int x = 0;
    private int y = 0;
}
```

```
Point origin = new Point();
```

Classes: Constructors Example

```
public class Point extends Object {
    private int x = 0;
    private int y = 0;
    // default constructor
    public Point() {
        // calls superclass constructor
         super();
    }
}
```

```
Point origin = new Point();
```

Classes: Constructors Example

```
public class Point {
    private int x = 0;
    private int y = 0;
    // custom constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
Point origin = new Point(); // Error
```

Classes: Constructors Example

```
public class Point {
    private int x = 0;
    private int y = 0;
    // custom constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
Point firstDot = new Point(10, 20);
```

Classes: Constructors Example

```
public class Point {
    private int x = 0;
    private int y = 0;
    // custom constructor
    public Point() {}
    public Point(int a, int b) {
        this();
    }
}
```

```
Point origin = new Point();
Point firstDot = new Point(10, 20);
```

Checkpoint

Class is a blueprint of an object

If not otherwise specified, class has a default no-arguments constructor

You can refer to the constructor of the class with **this()** keyword

You can refer to the constructor of the superclass with **super()** keyword

Object is an instance of a class

Object is instantiated with **new** keyword

Modifiers

Modifiers: Access modifiers

Modifier	**Class**	**Package**	**Subclass**	**World**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
(no modifier)	Y	Y	N	N
`private`	Y	N	N	N

Modifiers: Access modifiers

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
(no modifier)	Y	Y	N	N
`private`	Y	N	N	N

`Prefer private`

Modifiers: Access modifiers

Avoid possible errors

- **Use `private`**, unless you have a good reason not to
- Avoid `public` fields except for constants
  - `public` tend to link you to a particular implementation
  - limits your flexibility in changing your code

Modifiers: Non-access modifiers

`static`

Belongs to a class

`final`

Can’t be modified once declared

`abstract`

Incomplete implementation

`synchronized` and `volatile`

Used for threads

`transient`

Used in serialisation

Modifiers: Non-access modifiers

`static`

- Belongs to a class
- Is initialized once and shared by all instances of a class
- Can be applied to variables or methods

`final`

- Ensures that entities cannot be modified once declared
- Can be applied to variables, methods, or classes

`abstract`

- Incomplete implementation
- Can be applied to methods or classes

`synchronized` and **`volatile`**

- Used for threads

`transient`

- Indicate the JVM to skip the particular variable when serializing the object containing it

Modifiers: Static

```
public class Point {
    public int x = 1;
    public static int y = 2;
}
```

```
int xValue = Point.x; // Error
int yValue = Point.y;

Point origin = new Point();
int xValue = origin.x; // OK
```

Modifiers: Static

```java
public class Point {
    public int x = 1;
    public static int y = 2;
    public Point() {
        ++y;
    }
}
```

```java
Point origin = new Point();
Point origin1 = new Point();
Point origin2 = new Point();

int xValue = origin.x; // 5
```

Modifiers: Static methods

- Following combinations of instance and class variables and methods are allowed:
  - Instance methods can access directly
    - instance variables
    - instance methods directly.
  - Instance methods can access directly
    - class variables
    - class methods directly.
  - Class methods can access directly
    - class variables
    - class methods

Modifiers: Static methods

- Class methods cannot access directly:
    - Instance variables & methods
    - Instead **object reference** must be used
- Class methods cannot use the `this` keyword - **there is no instance for this to refer to**

Modifiers: Final constants

```
static final double ABSOLUTE_ZERO = 0; // constant

final Point startPoint = new Point(0, 0);
```

```
KelvinTemperature.ABSOLUTE_ZERO = -273.15; // Error

startPoint.setX(1);
```

Modifiers: Final

Methods

Cannot be overridden by any subclasses

The content of the method should not be changed by any outsider

Classes

Prevent the class from being subclassed

Cannot inherit from `final` class

Modifiers: Abstract

```
public abstract class Temperature {
    public double toCelsius(double t){
        return t - 273.15;
    }
}
```

```
Temperature temperatureKelvin = new Temperature(); // Error
```

Can contain both abstract methods as well normal methods

Modifiers: Abstract

```java
public abstract class Temperature {
    public abstract double toCelsius(double t);
}
```

```java
Temperature temperatureKelvin = new Temperature(); // Error
```

If a class contains abstract methods - class must be declared abstract

Checkpoint

There are 4 access modifiers: private, *no-modifier*, protected and public

Prefer to go from less permissive to more permissive access

Static variable – belongs to a class and can be changed

Final variable – cannot be changed after initialization, but objects state can

Abstract classes are meant to be subclassed

Abstract classes can contain both abstract methods as well normal methods

If a class contains abstract methods – class must be declared abstract

Nested Classes

Nested Classes

```
public class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

Prefer static nested classes

Nested Classes

- Why use nested classes?
  - It is a way of logically grouping classes that are only used in one place
  - It increases encapsulation
  - It can lead to more readable and maintainable code

Summary

Class is a blueprint of an object

Object is an instance of a class

There are 4 access modifiers: private, *no-modifier*, protected and public

Other modifiers: static, final, abstract …

Abstract classes are meant to be subclassed

Prefer static nested classes over non-static

Use nested classes as helpers that are not required for any other class

Home reading

https://docs.oracle.com/javase/tutorial/java/javaOO/index.html