Using Encapsulation and Inheritance

Dan Geabunea
SENIOR SOFTWARE DEVELOPER

@romaniancoder

Overview

Understand access modifiers and how to use them effectively

Encapsulation

Implement inheritance

Abstract classes

Demo: Using encapsulation and inheritance to model airspace routes

Access Modifiers

Access Modifiers

Specify if a given class or its members (fields and methods) can be accessed by other classes

Two Levels of Access Control

Top level (class level)

Member level

Class Level Modifiers

package-private (default)

A class is visible only to classes in the same package

public

A class is visible by the whole world

Top Level Modifier

No Explicit Modifier – Visible in Same Package

JetEngine.java

```
package pluralsight.oop;


// no modifier => package-private
class JetEngine {


}
```

Main.java

```
package pluralsight.oop;


public class Main {
    public static void main(String[] args) {
        // Can access Jet Engine, same package
        JetEngine je = new JetEngine();
    }
}
```

Top Level Modifier

No Explicit Modifier – Not Visible in Other Packages

JetEngine.java

```
package pluralsight.oop;

// no modifier => package-private
class JetEngine {

}
```

Main.java

```
package pluralsight.main;

public class Main {
    public static void main(String[] args) {
        // Can not access Jet Engine
        JetEngine je = new JetEngine();
    }
}
```

Top Level Modifier

Public Modifier – Visible to the World

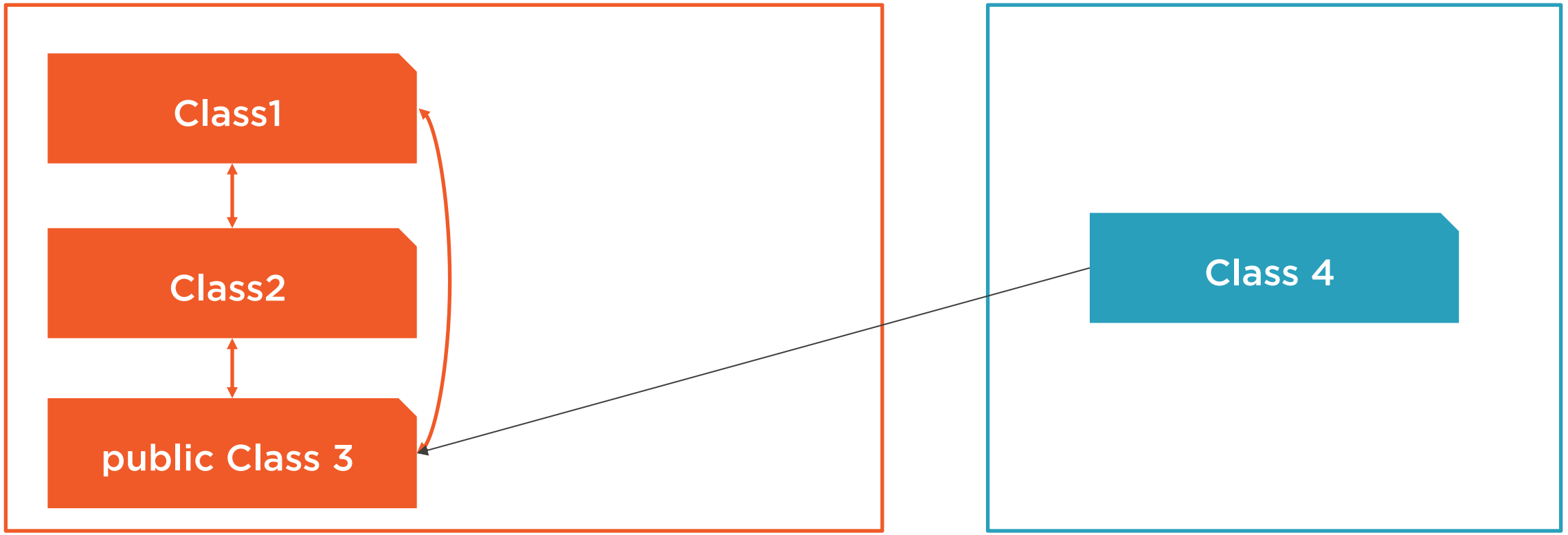
JetEngine.java

```
package pluralsight.oop;


// no modifier => package-private
public class JetEngine {


}
```

Main.java

```
package pluralsight.main;


public class Main {
    public static void main(String[] args) {
        // Can access public class Jet Engine
        JetEngine je = new JetEngine();
    }
}
```

Top Level Modifiers



Member Access Modifiers

Control access to a class fields and methods

Member Access Modifiers

Public, a member is visible to all the classes everywhere

Protected, a member is visible within its own package or by a subclass of its class in any other package

Package-private (no modifier), a member is visible only within its own package

Private, a member is visible only inside its own class

FlightDataRecorder.java

```
public class FlightDataRecorder {

    private String id;

    protected List<Entry> entries = new ArrayList<>();


    public FlightDataRecorder() {

        this.id = UUID.randomUUID().toString();

    }


    public void log(String msg) { entries.add(new Entry(LocalDateTime.now(), msg)); }


    public String getId() { return this.id; }

}
```

Access Level Table

	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

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

Encapsulation

Encapsulation

Grouping data with the methods that operate on that data within a class

Purpose of Encapsulation

Hide information

Hide internal workings

```
public class Aircraft {
    public int altitude;
    public int speed;
}


a.altitude = 0;
a.speed = 0;
```

Don’t Expose State

It can be exploited to use objects in ways that they were not meant to be used

Minimize errors from misuse

```
public class Aircraft {
        private int altitude, speed;
        public void land() {
                // Gradually decrease speed and altitude safely
        }
}
a.land();       // internal details are hidden, you only have access to public API
```

Don’t Expose More Than Necessary

Use the most restrictive access level that is suitable for your use case

Only expose the minimum functionality and data needed

It is not a good practice to use public fields

Getters / Setters

```
public class JetEngine {
    private int altitude;

    public int getAltitude() {
        return altitude;
    }

    public void setAltitude(int altitude) {
        if(altitude < 0 || altitude > 300){
            throw new IllegalArgumentException();
        }
        this.altitude = altitude;
    }
}
```

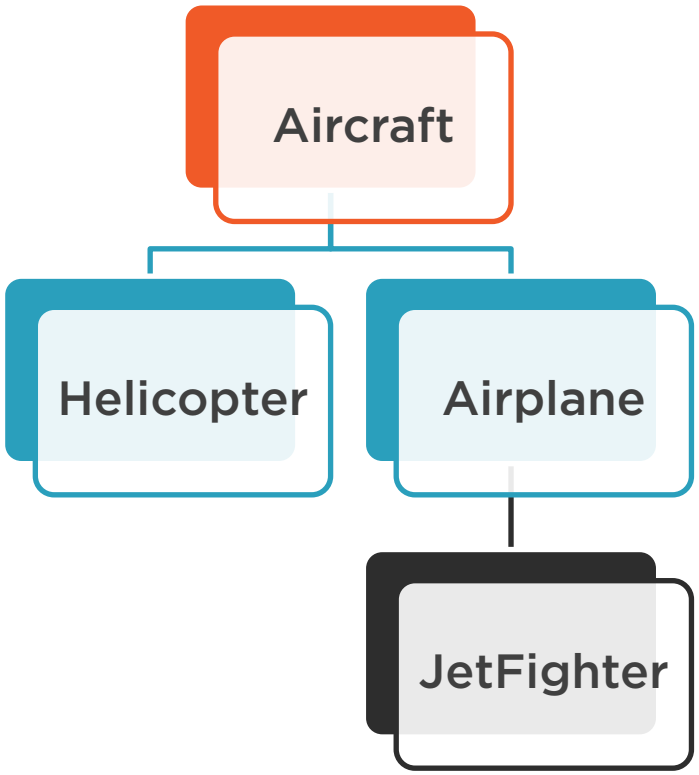
Inheritance

Inheritance

Define new classes based on existing classes.

A class can acquire the features and behaviors of another class and build upon them.

Class Hierarchies



We can inherit commonly used state and behavior from other classes

The “parent” class is called a base class or a superclass

The “child” class is called a subclass

A subclass extends a base class

In Java a class can extend a single class

If a class does not explicitly extend another class, then it implicitly extends Object

Basic Methods in Object Class

toString()

getClass()

hashCode()

equals()

clone()

Inheritance Example

Share Non-Private Members

```
public class Aircraft {
    private String modelId;
    protected int altitude, speed;

    public void land() {}

    public void takeOff() {}
}
```

```
public class Airplane extends Aircraft {
    // Can access all non private members
    // from the base superclass

    // Add another specific field
    private String wakeTurbulence;
}
```

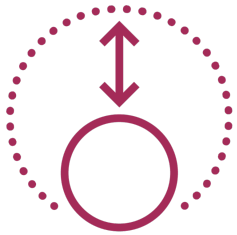
Subclass

The inherited fields can be used directly

The inherited methods can be used directly

You can create new instance fields in the subclass that do not exist in the super class

You can create new instance methods in the subclass that do not exist in the superclass



You can override existing instance methods

You can invoke the constructor of the superclass by using the “super” keyword

Method Overriding

The ability of a subclass to modify an existing method. The overriding method has the same name, number and type of parameters and return type

Method Overriding

```
public class Aircraft {
    public void land() {
        System.out.println("Aircraft landing");
    }
}
```

```
public class Helicopter extends Aircraft {
    @Override
    public void land() {
        System.out.println("Helicopter landing");
    }
}
```

```
Aircraft a = new Aircraft();

Aircraft h = new Helicopter();


a.land();   // Output => Aircraft landing

h.land();   // Output => Helicopter landing
```

Overriding Is a Form of Polymorphism

The version of a method that gets executed is determined at runtime, by the object that is used to invoke it

Prevent Method Overriding

Use the “final” modifier

```
public class Aircraft {

    public final void land() {

        System.out.println("Aircraft landing");

    }

}
```

```
public class Helicopter extends Aircraft {

    // Can not override final method

    @Override

    public void land() {

        System.out.println("Helicopter landing");

    }

}
```

The “super” Keyword

Access superclass members

Call superclass constructors

Access Superclass Members

```
public class Aircraft {
    public void land() {
        System.out.println("Aircraft landing");
    }
}
```

```
public class Helicopter extends Aircraft {
    @Override
    public void land() {
        super.land();
        System.out.println("Helicopter landing");
    }
}
```

Call Superclass Constructors

```
public class Aircraft {
    private String model;

    public Aircraft(String model) {
        this.model = model;
    }
}
```

```
public class Airplane extends Aircraft {
    private String wakeTurbulence;

    public Airplane(String model,
                    String wakeTurbulence) {
        super(model); // Call base constructor
        this.wakeTurbulence = wakeTurbulence;
    }
}
```

What if I don’t want anyone to mess with my class via inheritance

You

Prevent Inheritance

Use the “final” Modifier on the Class

```
public final class Aircraft {
    public void land() {
        System.out.println("Aircraft landing");
    }
}
```

```
// Not correct, can not extend final class
public class Helicopter extends Aircraft {
}
```

Abstract Classes and Methods

Abstract Class

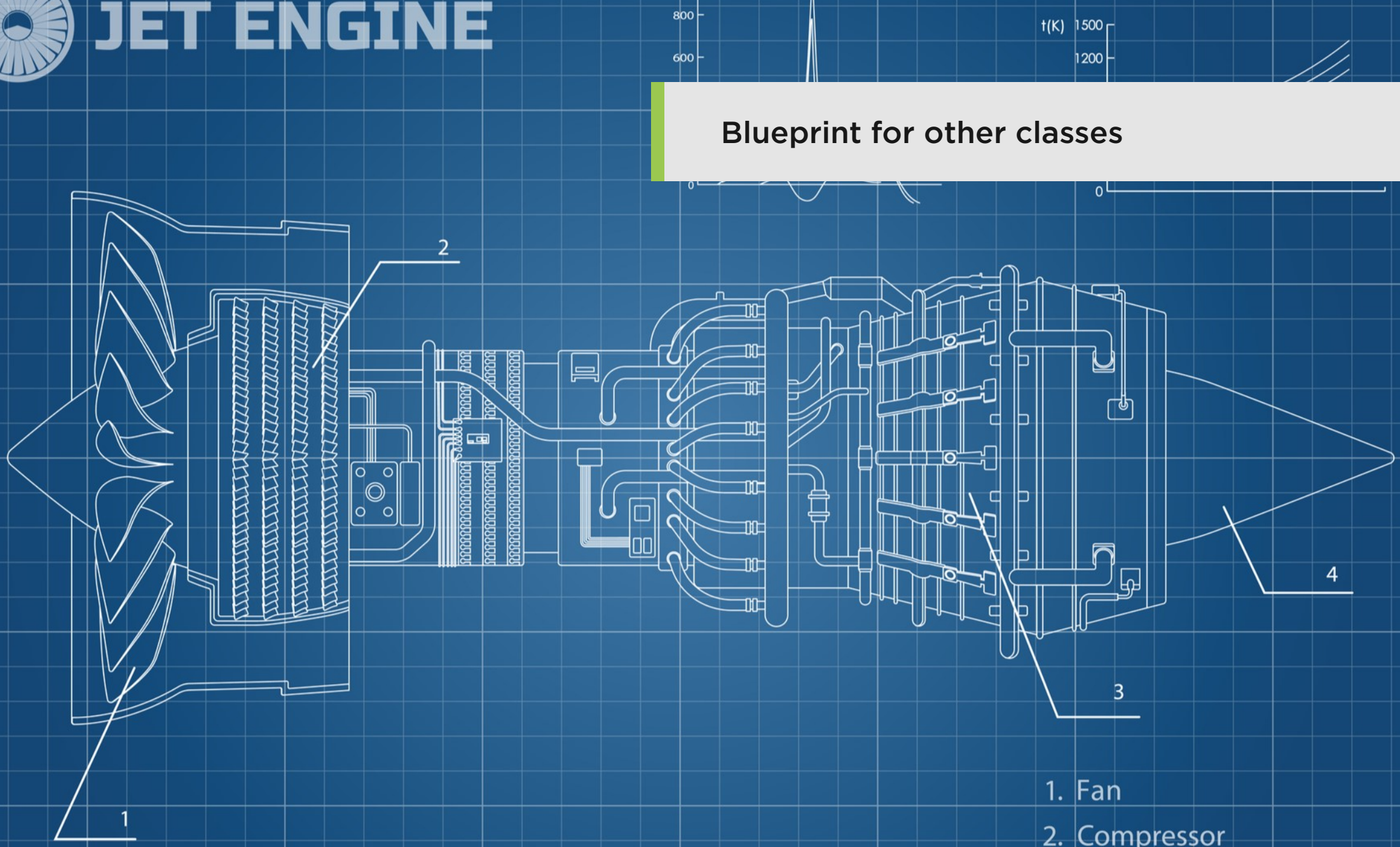
A class that can not be instantiated. It is meant to be extended and used as a baseline by more concrete classes

Abstract Method

A method that is declared without any implementation.
A subclass is responsible for overriding it.

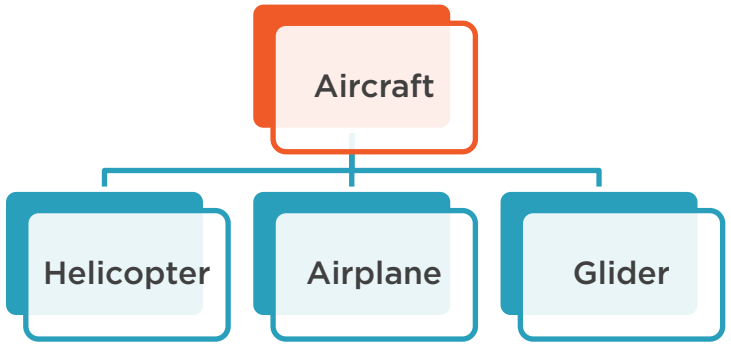
JET ENGINE

Blueprint for other classes



1. Fan
2. Compressor
3. Turbine

Class Hierarchies



It would be very unrealistic to expect the same landing procedure for all subclasses

However, they need to implement this action, but in their own way

An abstract class can be used to define a skeleton, or baseline. We abstract, but we leave freedom to subclasses to come with their own behavior in a consistent way.

Abstract Class

```
public abstract class Aircraft {
    private int altitude;

    // Too complex to be abstracted
    public abstract void land();

    public int getAltitude() {
        return this.altitude;
    }
}

Aircraft a = new Aircraft(); // Can't instantiate
```

Extending Abstract Classes

```
public class Helicopter extends Aircraft {
    @Override
    public void land() {
        // Complex logic that handles
        // helicopter landing
    }
}
Aircraft h = new Helicopter();
h.land();
int alt = h.getAltitude();
```

```
public class Glider extends Aircraft {
    @Override
    public void land() {
        // Complex logic that handles
        // glider landing
    }
}
Glider g = new Glider();
g.land();
int alt = g.getAltitude();
```

When to Use Abstract Classes

Create a template for future classes, but let them provide the actual implementation

Share some functionality with future subclasses

Implement Template Method design pattern

Template Method

Design the Skeleton of an Algorithm

```
public abstract class Aircraft {
    public final void takeOff(){
        this.checkSystems();
        this.getClearance();
        this.implementCustomActions();
    }
    // A subclass needs to implement all these abstract methods
    protected abstract void checkSystems();
    protected abstract void getClearance();
    protected abstract void implementCustomActions();
}
```

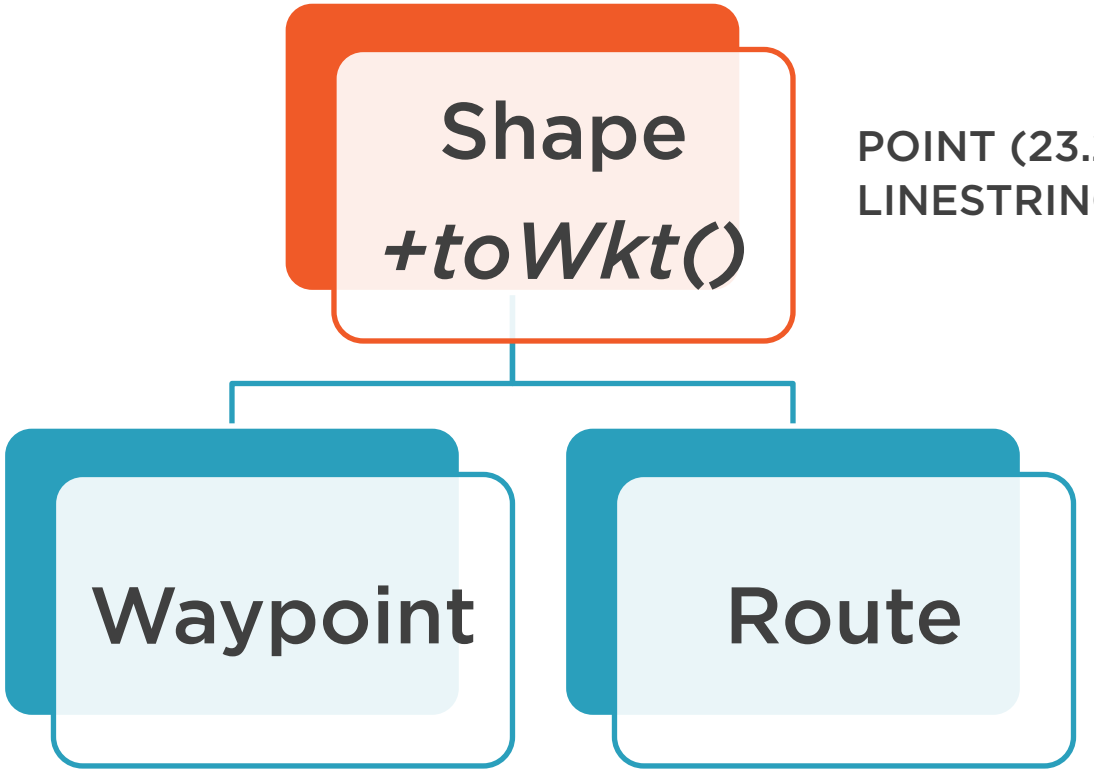
Abstract Clases

- Must be declared using the “abstract” keyword
- Can contain abstract and non-abstract methods
- Can have constructors
- Can not be instantiated

Demo

Demo: Using encapsulation and inheritance to model the display of airspace routes

Airspace



POINT (23.2 44.5)
LINESTRING (2.3 41.3, 2.4 41.5, 2.7 41.7)

Summary

Access modifiers can help us to hide away implementation details and protect our classes from misuse

Inheritance is a great way to share functionality in a group of related classes

We can use method overriding to give a more specific implementation than the one defined in a base class

We can prevent inheritance and overriding by using the “final” keyword

Abstract can not be instantiated

Abstract classes can be used as a template by other classes

Up Next:
Understanding Interfaces and Polymorphism