Java Object Oriented Approach (Java SE 11 Developer Certification 1z0-819)

INTRODUCTION TO OOP: CLASSES AND OBJECTS

Dan Geabunea
SENIOR SOFTWARE DEVELOPER

@romaniancoder

Objects are all around us

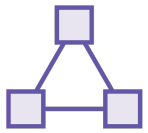
Our brains naturally see the world as an organized collection of various objects

OOP

Is a programming paradigm that organizes software around objects. In most cases, this is the most natural and pragmatic way of modelling the real world.

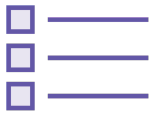
Course Overview

Fundamental concepts of OOP



Abstraction, encapsulation, inheritance and polymorphism

Static members and classes



Enumerations and nested classes

Prepare for the Java SE 11 Developer Certification 1z0-819

Learn how to use OOP effectively to solve real business problems, and gain all the OOP knowledge needed for the Java 11 SE Developer Certification

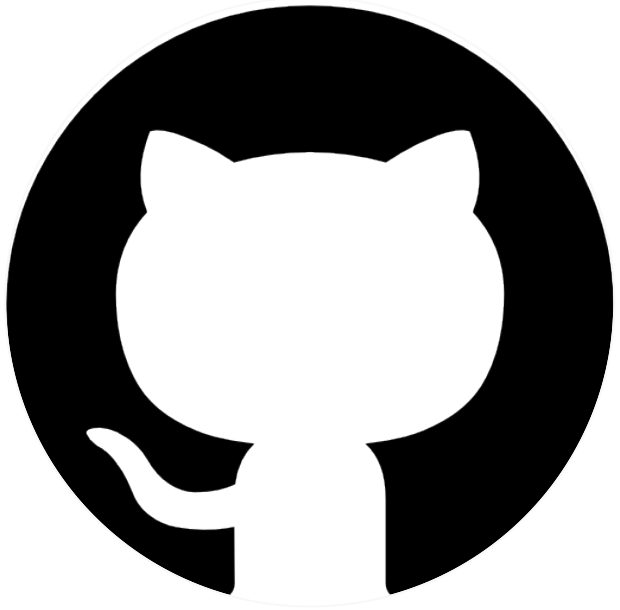
Configuring the Development Environment

Requirements

Java 11 SDK

Maven (3.6)

IntelliJ



Source Code

Available on GitHub and included as course assets

https://github.com/dangeabunea/pluralsight-java11-object-oriented-approach

main · 1 branch · 0 tags · Go to file · Add file · Code

dangeabunea removed idea and target folders — 4a82863 · 3 days ago · 4 commits

1-classes-and-objects	removed idea and target folders	3 days ago
.gitignore	Created before/after dor m1	3 days ago
README.md	Work on first module demo	3 days ago

before	removed idea and target folders	3 days ago
complete	Created before/after dor m1	3 days ago

Overview

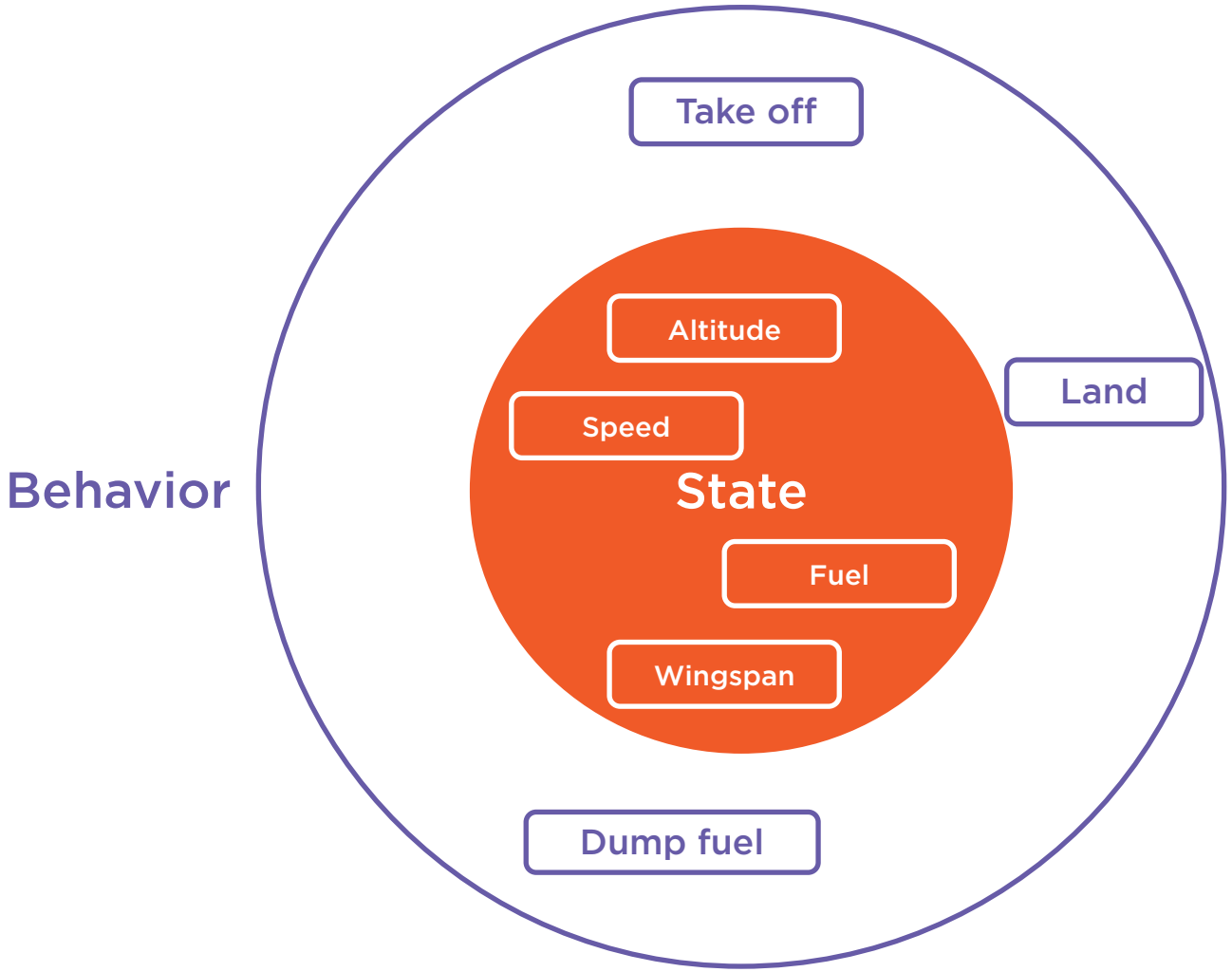
- Understand objects
- Define classes
- Use constructors & initializers
- Organize classes with packages
- Understand garbage collection

Objects

Object

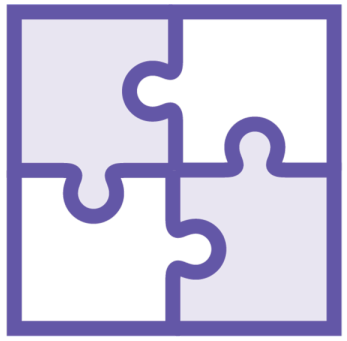
A software construct that models real world concepts

Aircraft State and Behavior

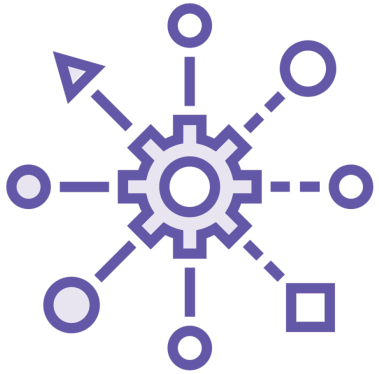


Objects should remain in control of how the outside world can use them

Benefits



Modularity



Code re-use

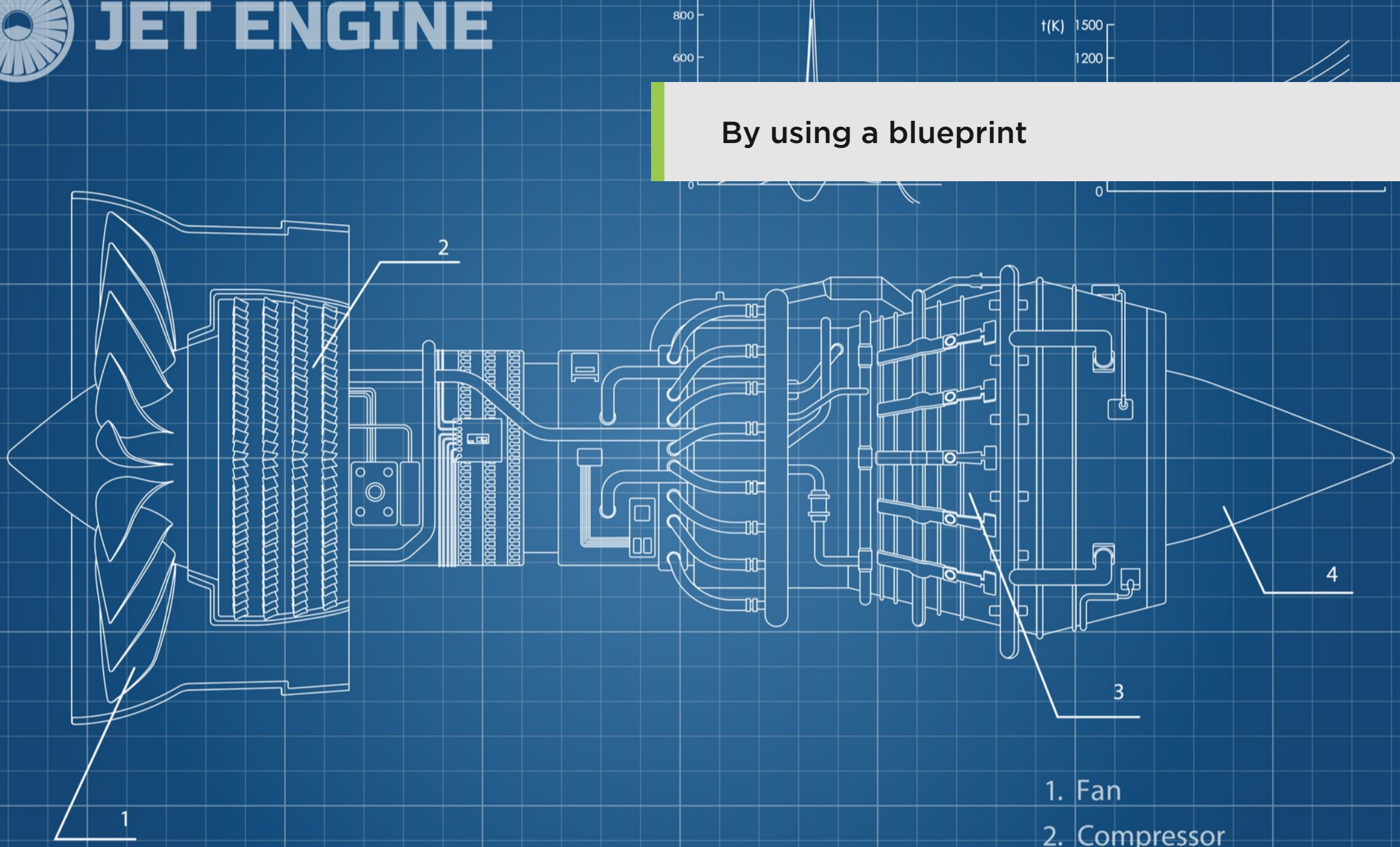
Information hiding

Classes

How do you build a jet engine?

JET ENGINE

By using a blueprint



1. Fan
2. Compressor
3. Turbine

Class

A prototype from which objects can be created/instantiated. It contains all the properties and methods that are common to all objects of that type.

JetEngine.java

```
class JetEngine {
    // fields (state)
    String model;
    int fanSpeed;
        int maxFanSpeed;
    int thrust;


    // methods (behavior)
    void start() { }
    void stop() { }
}
```

JetEngine.java

```java
public class JetEngine {
    // fields (state)
    String model;
    int fanSpeed;
        int maxFanSpeed;
    int thrust;

    // methods (behavior)
    void start() { }
    void stop() { }
}
```

JetEngine.java

```
public class JetEngine extends Engine implements PropulsionSystem {
    // fields (state)
    String model;
    int fanSpeed;
        int maxFanSpeed;
    int thrust;

    // methods (behavior)
    void start() { }
    void stop() { }
}
```

Class Anatomy

Class access modifier	Class keyword	Class name
Superclass and/or interfaces	Fields	Methods

Understanding Constructors

Constructor

A special method used to initialize objects from a class

Constructor Properties

- Constructors are like methods except they use the name of the class and have no return type
- A default, no-argument constructor is available even if you don't declare any
- You can declare multiple overloaded constructors
- They have access modifiers

Default Constructor

```
class JetEngine {
        String model;
        int fanSpeed;
        int maxFanSpeed;
        int thrust;
        void start() { }
        void stop() { }
}
JetEngine trent = new JetEngine();
trent.start();
```

Instantiate a Class

```
class JetEngine {
        String model;                   // null
        int fanSpeed;                   // 0
        int maxFanSpeed;                        // 0
        int thrust;                     // 0
        void start() { }
        void stop() { }
}
JetEngine trent = new JetEngine();
trent.start();
```

Explicit Constructor

```
class JetEngine {
        String model;
        int fanSpeed;
        int maxFanSpeed;
        int thrust;

        JetEngine(String model) { this.model = model; }
}
JetEngine trent = new JetEngine("Trent 800");
System.out.println(trent.model);     // "Trent 800"
```

Explicit Constructor

```
class JetEngine {
        String model;
        int fanSpeed;
        int maxFanSpeed;
        int thrust;

        JetEngine(String model) { this.model = model; }
}
JetEngine trent = new JetEngine();          // Won't work
```

Explicit No-argument Constructor

```
class JetEngine {
        String model;
        int fanSpeed;
        int maxFanSpeed;
        int thrust;

        JetEngine() { }
        JetEngine(String model) { this.model = model; }
}
JetEngine trent = new JetEngine();
```

Constructors can be overloaded and reused by other constructors

Overloaded Constructors

```
class JetEngine {
        String model;
        int maxFanSpeed;

        JetEngine(String model) { this.model = model; }
        JetEngine(String model, int maxFanSpeed) {
                this(model);        // needs to be the first line
                this.maxFanSpeed = maxFanSpeed;
        }
}
JetEngine trent = new JetEngine(“Trent 800”, 60000);
```

Overloaded Constructors: Order of Execution

```
class JetEngine {
        JetEngine() { System.out.println("no arg"); }
    JetEngine(String model) { this(); System.out.println("model"); }
    JetEngine(String model, int maxFanSpeed) {
                this(model); System.out.println("model, maxFanSpeed");
        }
}
JetEngine trent = new JetEngine(“Trent 800”, 60000);
```

no arg

model

model, maxFanSpeed

Private Constructors

```
class JetEngine {
        int maxFanSpeed;

        private JetEngine(String model) { this.model = model; }
        JetEngine(String model, int maxFanSpeed) {
                this(model);        // needs to be the first line
                this.maxFanSpeed = maxFanSpeed;
        }
}
JetEngine trent = new JetEngine("Trent 800", 60000);      // Ok
JetEngine trent = new JetEngine("Trent 800");             // Compilation Error
```

Overloaded constructors act like overloaded methods. The compiler differentiates them by the number of arguments and their type

Overloaded Constructors

Need to Have Different Signatures

```
class JetEngine {
        int maxFanSpeed;
        int maxThrust;


        // Not valid, compiler error
        JetEngine(int maxFanSpeed) { this.maxFanSpeed = maxFanSpeed; }
        JetEngine(int maxThrust) { this.maxThrust = maxThrust; }
}
```

Initializer Block

Code that is executed whenever an instance is created. It has no data type, no associated name and is placed outside of any method.

Initializer Types

Static

Used for initializing static fields

Non-Static

Used for initializing instance fields

Instance Initializer

```
class JetEngine {
        String model; int maxThrust;


        {
                this.model = “”;
                System.out.println(“Initializer called”);
        }


        JetEngine(int maxFanSpeed) { this.maxFanSpeed = maxFanSpeed; }
}
JetEngine trent800 = new JetEngine(6000);
```

Instantiating Objects

Instantiating Objects

```
JetEngine trent800 = new JetEngine("Trent 800", 6000);

Aircraft boeing = new Aircraft("Boeing 737");
```

1. **Declaration**, associate a variable name with the object type
2. **Instantiation**, the ‘new’ keyword creates a new object
3. **Initialization**, call to constructor

Declaring a Variable to Refer to an Object

```
JetEngine trent800;
```

This notifies the compiler that you will use the variable called ‘trent800’ to refer to data who has the ‘JetEngine’ type

The value will be determined after instantiation. Simply declaring a reference variable does not create an object

Instantiating an Object

```
JetEngine trent800 = new JetEngine("Trent 800", 6000);
```

The ‘new’ operator allocates memory for the ‘JetEngine’ object and returns a reference to that memory location

Then, it invokes the constructor

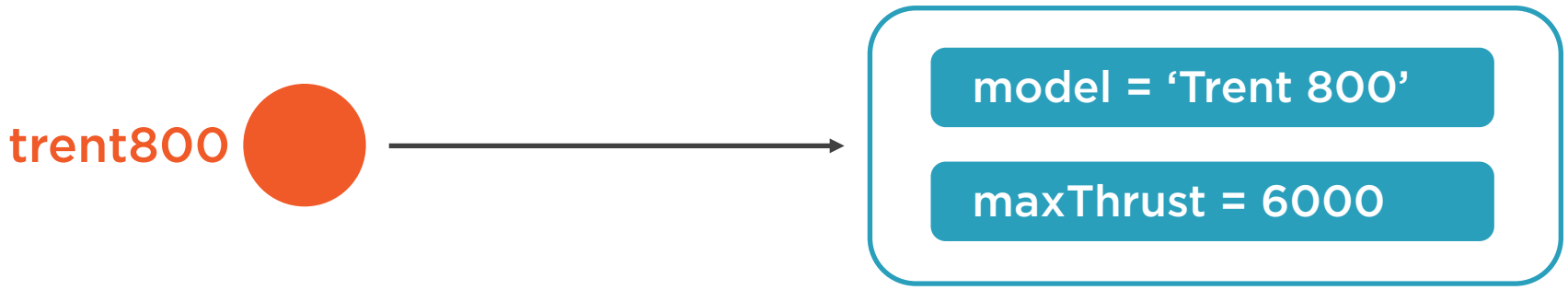
Initializing an Object

```
JetEngine(String model, int maxFanSpeed) {
        this.model = model;
        this.maxFanSpeed = maxFanSpeed;
}
```

The constructor takes care of object initialization

```
JetEngine trent800 = new JetEngine("Trent 800", 6000);
```

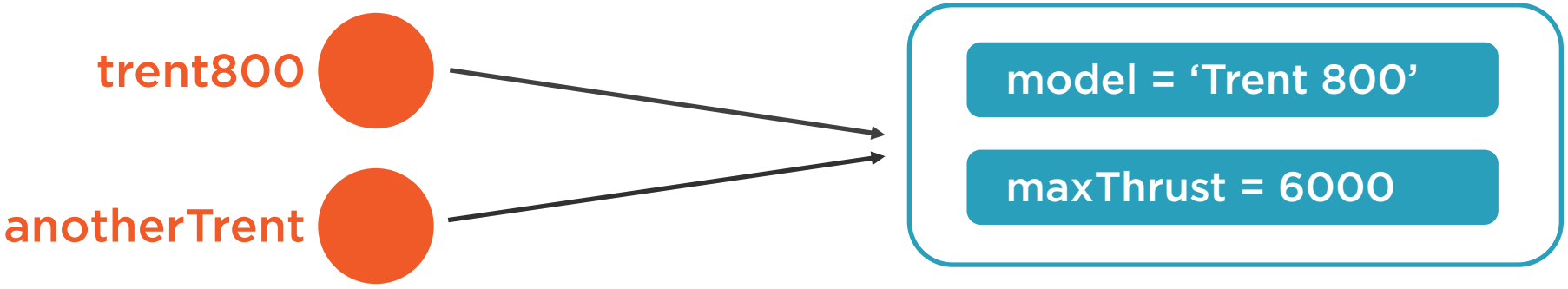
Creating Objects Process



```
JetEngine trent800 = new JetEngine("Trent 800", 6000);

JetEngine anotherTrent = trent800;
```

Multiple References to the Same Object

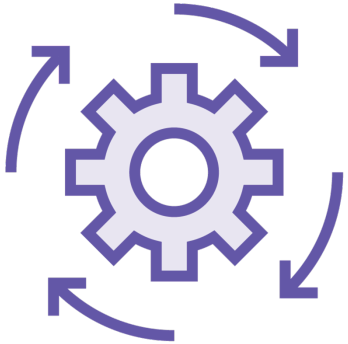


Destroying Objects

Garbage Collection

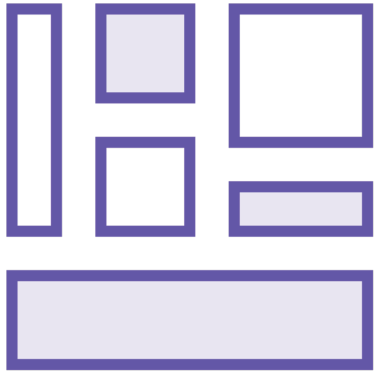
Process that finds and deletes unused objects from the memory heap

Garbage Collection



Automatic

Memory is freed up automatically



Customizable

GC parameters are highly customizable

Nondeterministic

You have little control on the process

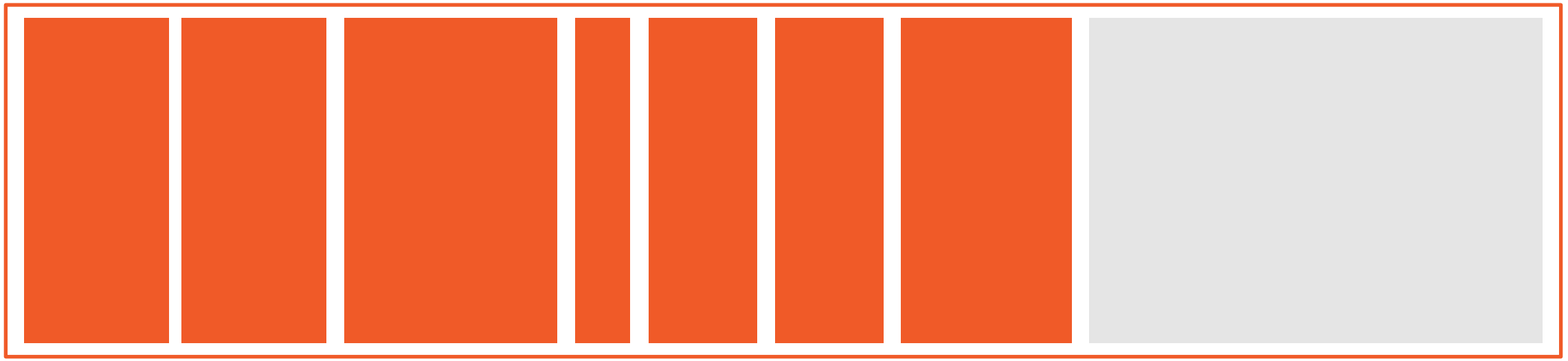
Basic GC Process

Mark – identify which objects are in use and which are not

Delete – remove unreferenced objects to free memory

Compact – optional step, compact remaining referenced objects

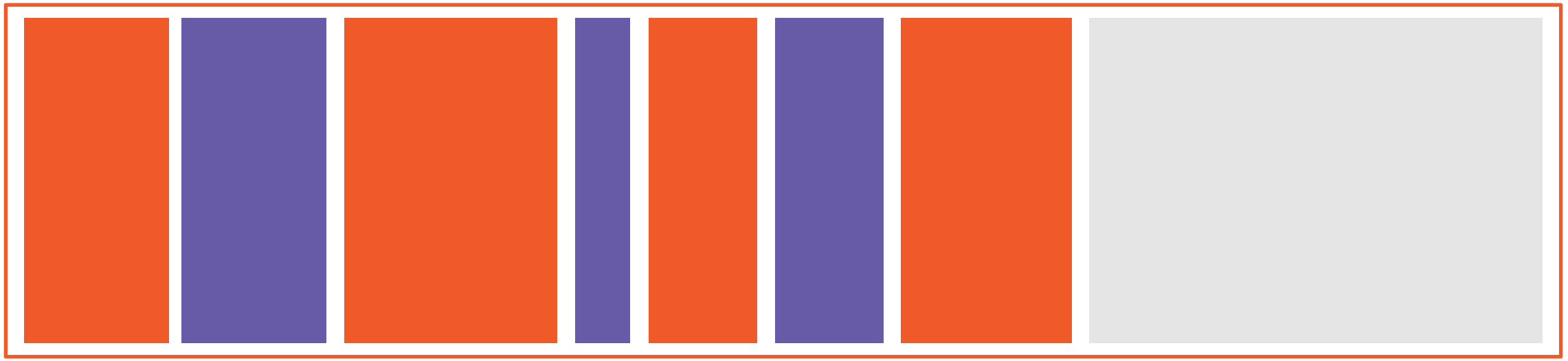
Basic GC Process - Heap



Objects

Free Space

Basic GC Process - Mark

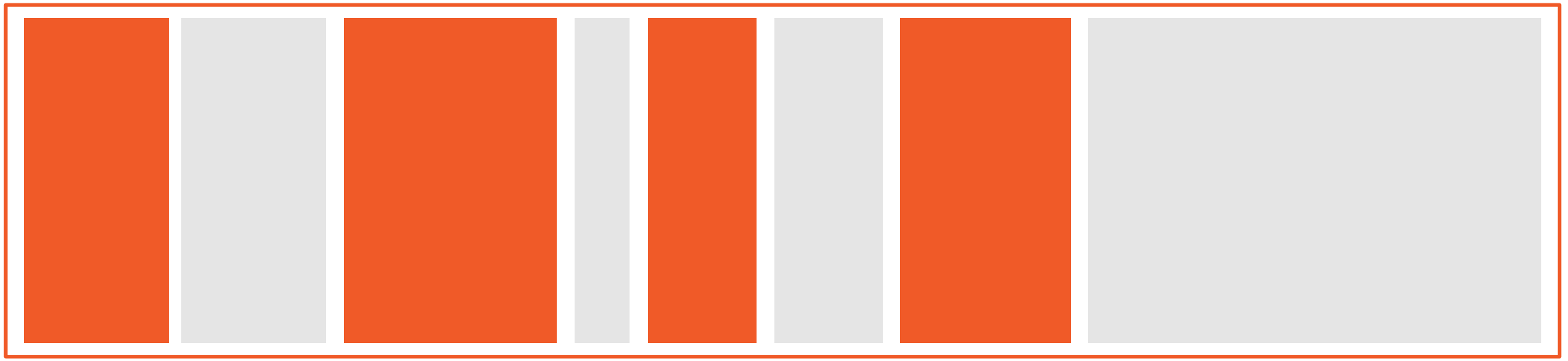


Objects

Free Space

Unreferenced

Basic GC Process - Delete



Objects

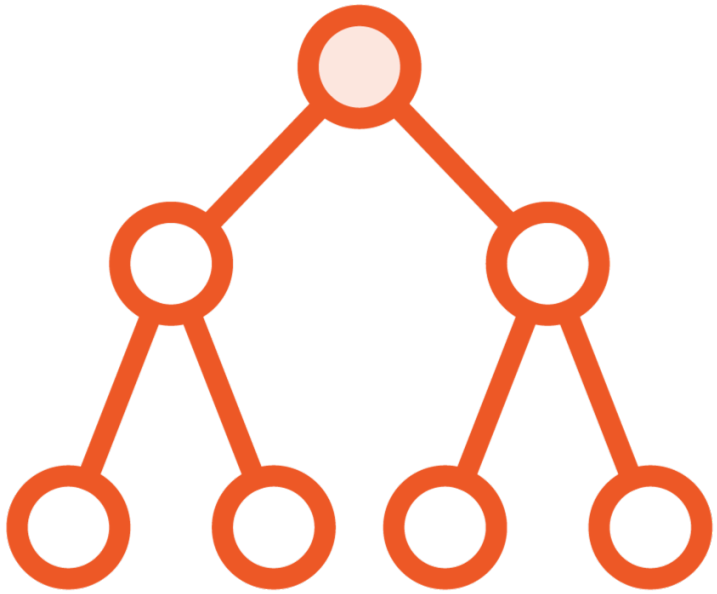
Free Space

Basic GC Process - Compact

Objects

Free Space

GC Roots



Local variables

Active Java threads

Static variables

Understanding the Java Virtual Machine: Memory Management

Kevin Jones

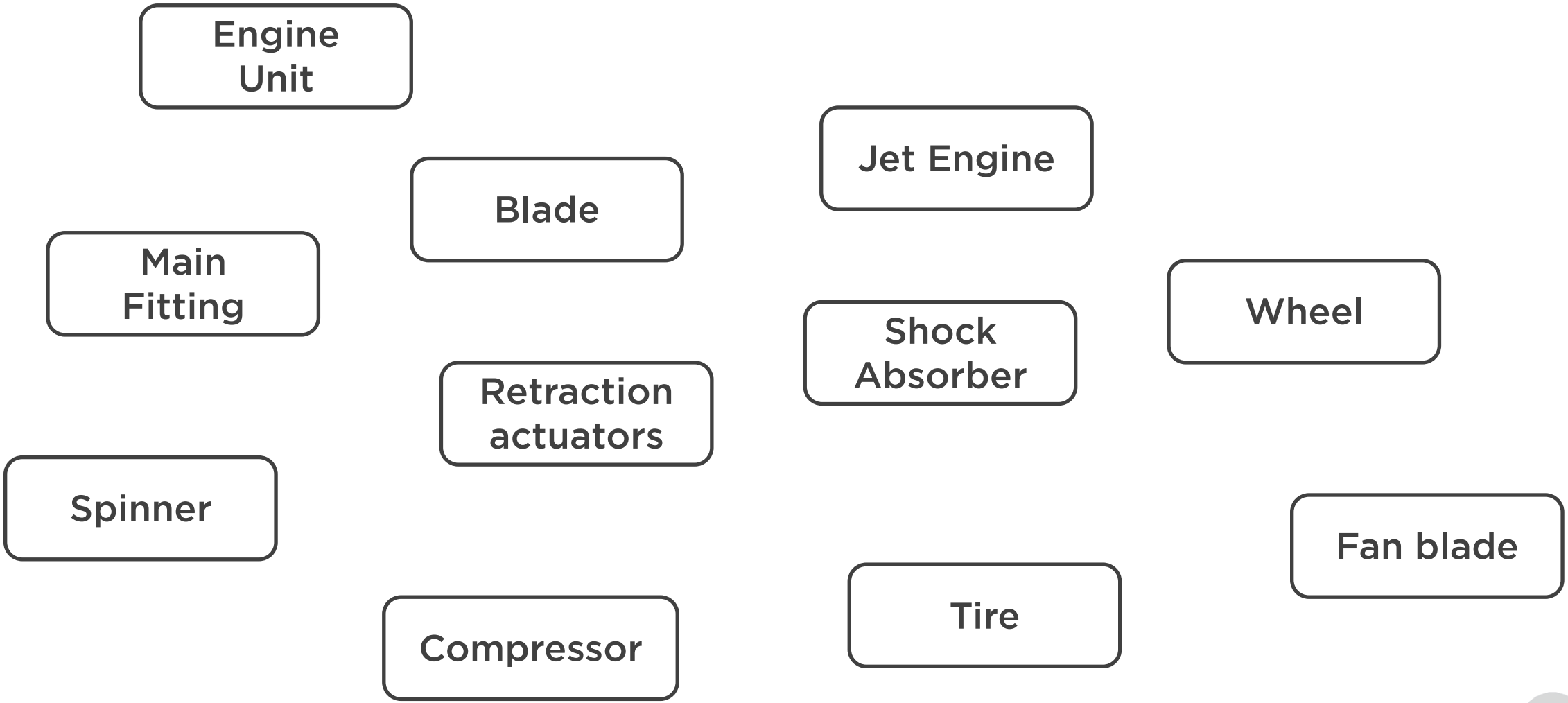
Organizing Classes with Packages

Package

A namespace that organizes a group of related classes or interfaces

You can think about packages as folders on your computer

Unorganized Code



Organized Code

com.aircraft.system.engine

- Engine Unit
- Blade
- Jet Engine
- Fan blade
- Compressor

com.aircraft.system.landing

- Retraction actuators
- Spinner
- Main Fitting
- Tire
- Wheel
- Shock Absorber

Package Naming Conventions

Package names are written in lower case

Companies usually use their reverse domain name as a prefix (com.pluralsight.videocourse)

Packages in the Java API start with java or javax

main.java

```
package com.aircraftfactory.logistics;


import java.time.LocalDateTime;
import com.aircraftfactory.shipping;


public class Planner {
        // Code goes here
}
```

Demo

Defining & Instantiating Classes

- Define a class to model a flight plan
- Add overloaded constructors
- Understand constructor execution order

Summary

OOP is a paradigm that organizes software around objects. Most times it is the most efficient way of modelling complex business needs

Classes are a blueprint, from which multiple objects can be instantiated

Constructors are a special method that allows us to initialize objects

Packages are namespaces that organize code based on its purpose or responsibility

Objects are destroyed automatically, by a process called garbage collection

Up Next:
Modelling State and Behavior