Object-Oriented Programming Part 1

Java Accelerator 7

Lesson 1.2

In this lesson, we compare and contrast the OO approach of Java and other languages.

Learning Objectives

- Design Java classes.

- Instantiate and reference common Java types.

Object Basics

Properties of an Object-Oriented Language

- Everything is an object.

- A program is a collection of objects telling each other what to do by sending messages (in Java's case, these messages are **method calls**).

- Each object can be made up of other objects (called **composition** in Java).

- Every object has a type.

- All objects of a particular type can receive the same messages (in Java this means that they all have the same methods).

Everything Is an Object

- What does this mean?
- Is Java completely object-oriented?
- What about primitive types?
- Does the fact that primitive types are not objects adversely affect anything?

What Is a Program?

A **program** is a collection of objects telling each other what to do by sending messages. In Java’s case, these messages are **method calls**.

More About Objects

Composition in Java

Each object can be made up of other objects (called composition in Java). For example:

A house HAS (or contains) a stove, refrigerator, TV, washing machine, sink, and so on.

All of these are also objects.

Has-A Relationships

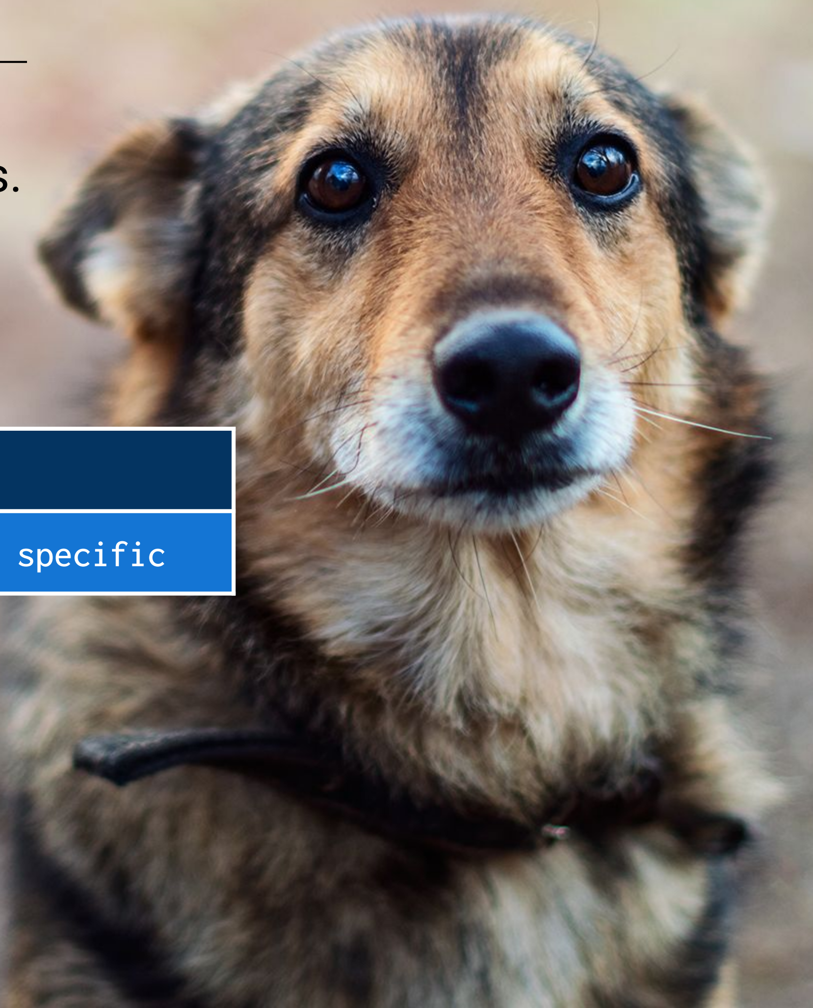
A car HAS AN engine.

Every Object has a Type

We've discussed data types for primitives: numbers, text, and Booleans.

Classes and objects are no different: each class and object has a type.

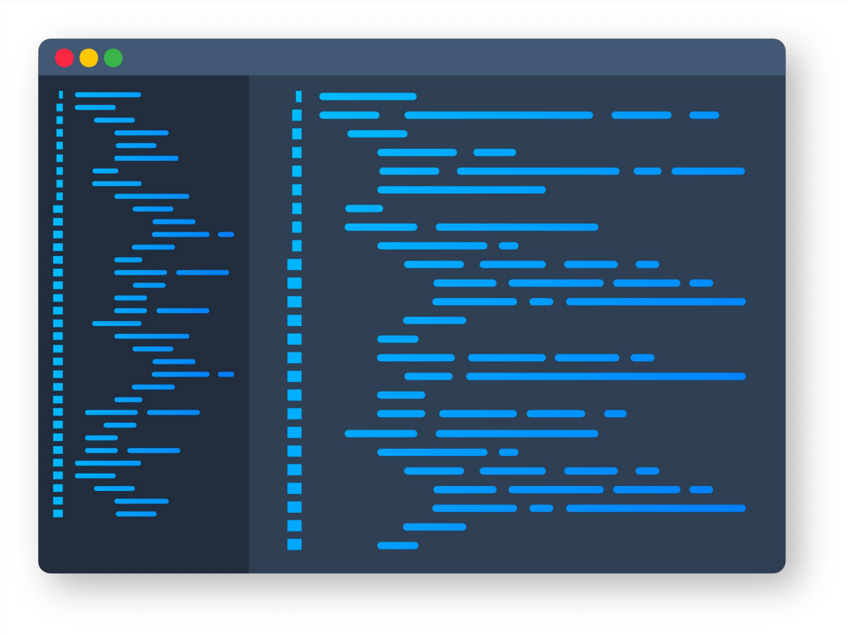
animal ->	mammal ->	dog
general ->	more specific ->	even more specific



All objects of a particular type can receive the same messages. In Java, this means that they all have the same methods.

Methods

This means all objects of the same type resemble each other; they have the same features. This makes sense because all objects of a given type are created from the same code.



Simple Definition of an Object:

An object has state, behavior, and identity.

—*Grady Booch,*
Software engineer

State, Behavior, Identity

State / Properties

Each car has a property that is its color—all cars have a color.

Property Yellow

Property Blue

State / Properties

But each car can have a different color (some blue, some yellow, some black).
In other words, they can all have different color values—we call this **state**.

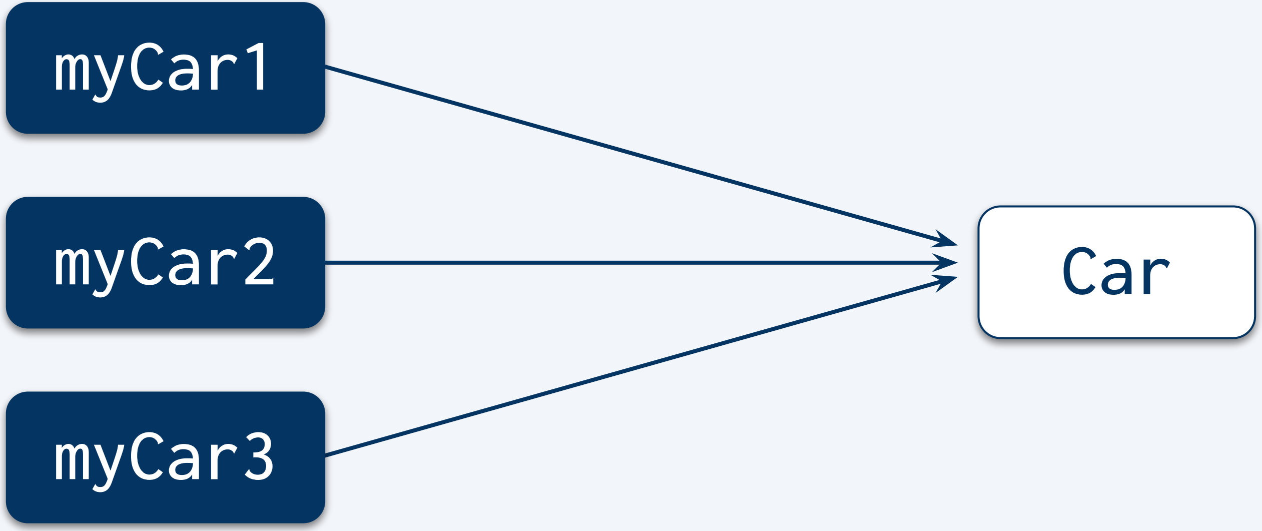
We can change the state of a car by painting it a different color, at which time the value of the color property would change. This changes the state of the car!

Behavior refers to the methods or capabilities of the object.

For example, a car can open its doors or move forward.

Identity

You can distinguish each object from all of the other objects in a program.

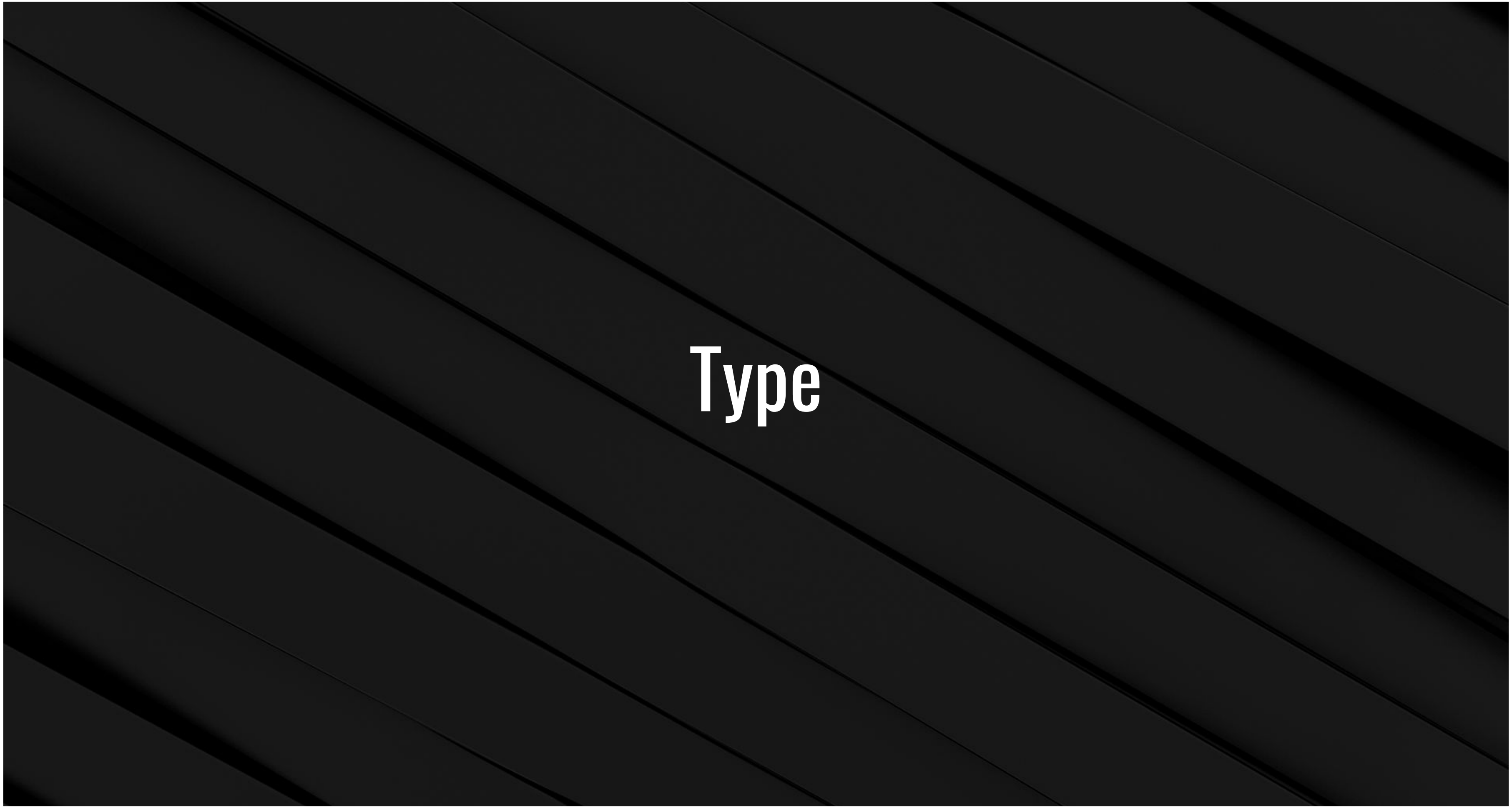


Instructor Demonstration

Create a Simple Class

Activity: Create a Student Class

Suggested Time:



Type

Every Object Has a Type

- Every time we define a new Java class, we define a new type.
- This concept will become more clear when we start creating new classes.

Public Interface and Private Implementation

Access Modifiers

In Java, access modifiers are used to define the accessibility of classes and also methods and variables within classes.

The access modifier for a class can be either **public** or **default**.

Access modifiers for class methods and variables include **public**, **private**, **default** and **protected**.

About Public Interface and Private Implementation

- Every class should have a well-defined way to interact with the world. Generically, we call this its **public interface**.

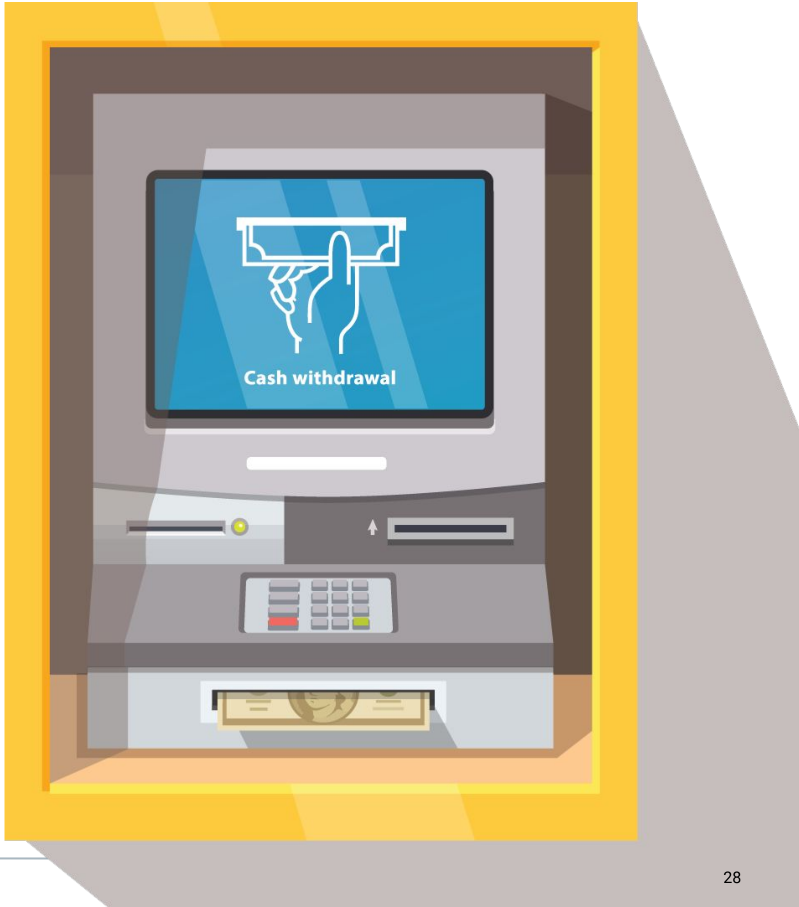
- The implementation of the class (HOW it interacts) should be hidden from the outside world—we call this **private implementation**.

- This allows us to separate the "what" from the "how." Clients of an object should only be concerned with what it can do, not how that's done.

- This allows the person implementing the class to change the implementation at their discretion.

Public Interface and Private Implementation Examples

- ATM machines
- Amazon
- Drive-through pharmacy



Encapsulation

Classes provide us with two key features of **encapsulation**.

- Bundling together data and the code that works with that data.
- Protecting the data in a class so that **only** the methods of the class can modify the data. We call this **data-hiding**.

How are an ATM, Amazon, and a drive-through pharmacy cohesive/encapsulated?

What if they weren't cohesive/encapsulated?

Data Hiding

Data hiding is closely related to encapsulation.
It simply means that the internal state of the object is hidden from the rest of the world.

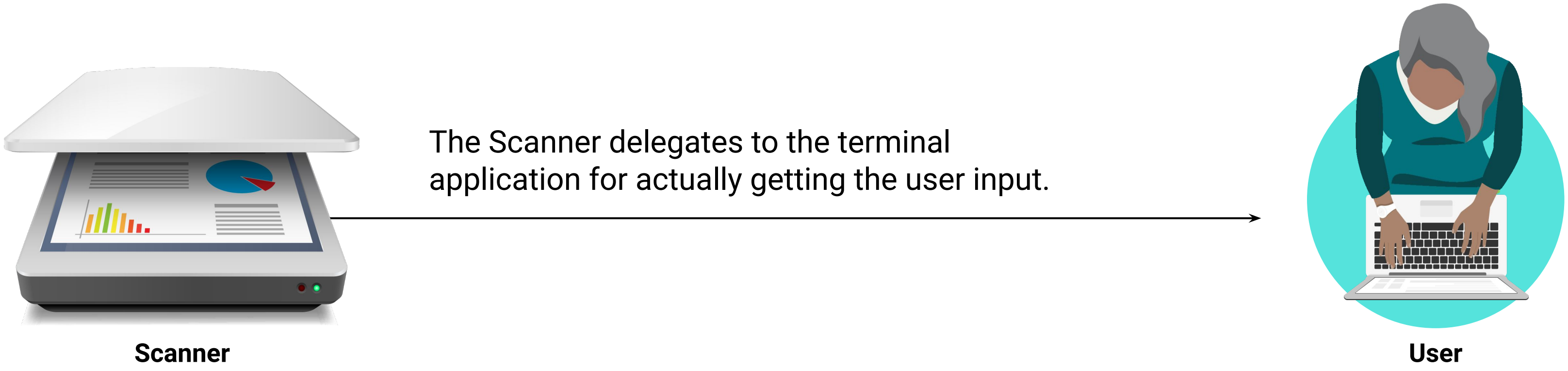
The internal state of the object is part of the private implementation.

Delegation

Delegation

Delegation goes along with encapsulation. In many cases, the private implementation of a class will delegate work to an existing class rather than reinvent the wheel.

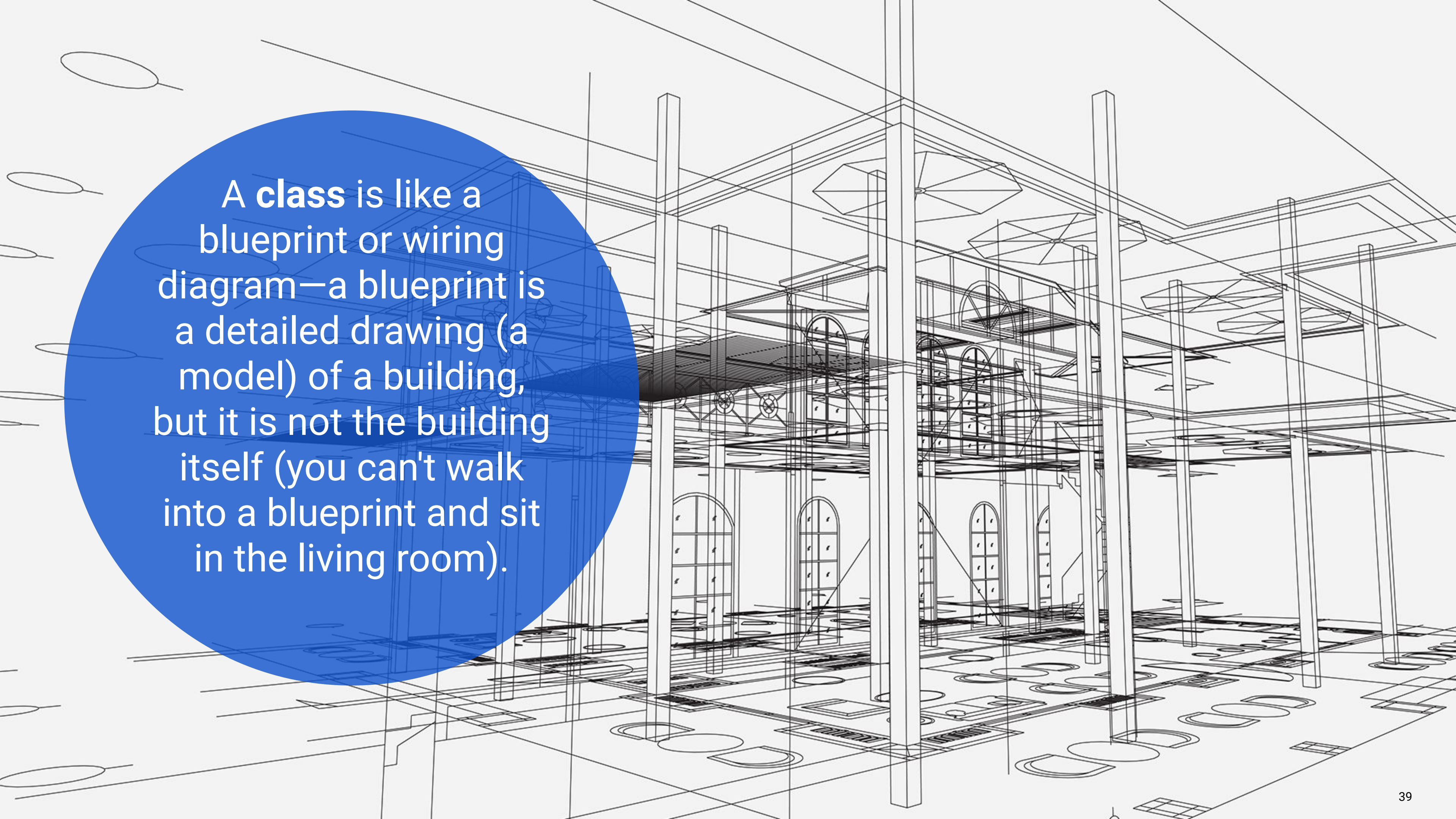
For example, we delegate to the Scanner for reading from the console and reading from files.



The Scanner delegates to the terminal application for actually getting the user input.

What are some real-world examples of delegation?

Classes and Objects



A **class** is like a blueprint or wiring diagram—a blueprint is a detailed drawing (a model) of a building, but it is not the building itself (you can't walk into a blueprint and sit in the living room).

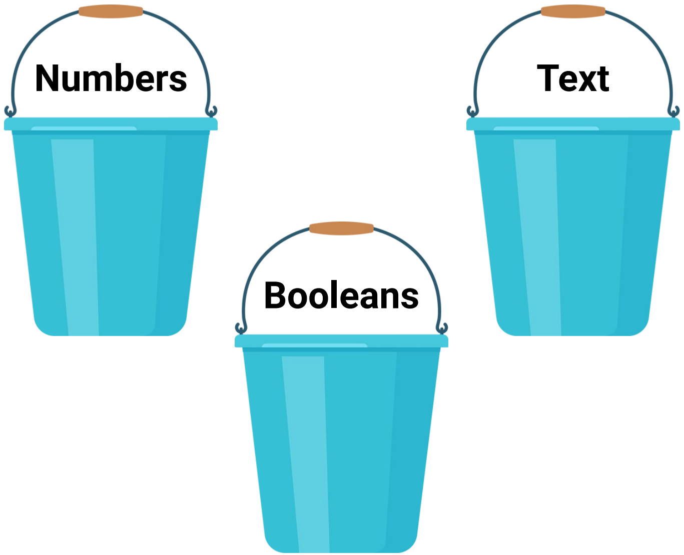
The same goes for a wiring diagram; it is a model. You can't plug your vacuum into a wiring diagram! Similarly, the class is the definition (or drawing) of an object.

Just like you have to build a house from a blueprint, you must **instantiate** an object from a class definition.

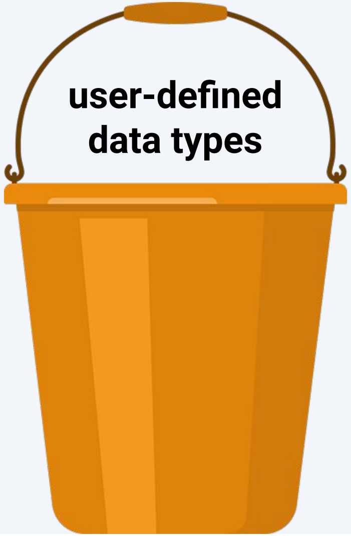
Creating New Types

About New Types

We have already covered type. As of now, we know three main categories of native types:



Another bucket, or category, of data type is called user-defined data types.
This bucket is very large; every time we create a new class, we create a new type.



These types (classes) are made up of:

- **Properties** (also called fields)
- **Methods** (also called behaviors)

Properties, Getters, and Setters

Properties represent some feature of a class, like color, name, or age.

Classes are models of something.

To implement data hiding and prevent direct manipulation of property values, we use **getters** and **setters**.

Constructors

About Constructors

A **constructor** is a special method that is called when you instantiate a class.

Constructors **initialize** (set the internal state) of the object being created from the class.

Restrictions on Constructors

A constructor must have the same name as the class that contains it.
For example, if we have a class called `Lion`, the constructor must be called `Lion()`.

Like general-purpose methods, constructors can have zero or more parameters.

Unlike general-purpose methods, constructors have no return type (not even void).

A class can have more than one constructor.

If you don't supply a constructor for your class, the compiler will supply one (called the default constructor). This default constructor has no parameters and contains no code in its body.

Time to Code

Lion Class

Suggested Time:

`this` Keyword

`this` Keyword Example

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

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
 }
```

`static` Keyword

Properties and methods marked with the `static` keyword are associated with the CLASS and not with any particular instance of the class.

About the `static` Keyword

Because static properties and methods are associated with the class:

01

There is only one copy of a static property or method.

02

Static methods can only directly access other static methods in a class.

03

Static methods can be accessed without creating an instance of the class.

Time to Code

Review Lion Class

Suggested Time:

References and Memory Management

Instantiating Objects in Java

Constructors are special methods that are called when we instantiate an object from a class. The constructor is used to initialize the object. Initialization means setting values for the properties of the object so that the object is in a known state.

- We indicate that we want to instantiate a new object by using the new keyword.
- When the new keyword and a constructor are used together, the JVM creates a new object in a part of memory called the **heap** and gives us a **reference** to that object.

Instantiation and Use Example

```
Scanner myScanner = new Scanner(System.in);
System.out.println(“What’s the lion’s name?”);
String newName = myScanner.nextLine();

Lion bigCat = new Lion(newName, 3, 200);
String bigCatName = bigCat.getName();
bigCat.nap();
bigCat.roar();
```

Memory, Variables, and Object References

Stack and Heap

- Stack memory is limited, so stack variables only exist for the duration of the code block that they're a part of.
- Local primitive values live on the stack.
- The reference variable (the part that holds the location of the object on the heap) also lives on the stack.
- Objects created with the new operator live on the heap, and we use references to access them.

In Java, like other languages, developers are not responsible for explicitly reserving and releasing memory resources.

This means that we can just create objects (using the new operator) whenever we need them.

Memory Management and Garbage Collection

The process of reclaiming memory is called garbage collection, and the component responsible for the process is called the **garbage collector**.

Objects on the heap are eligible for **garbage collection** when there are no more references pointing to them.

We can explicitly stop a reference from pointing to an object by setting the reference to null, as shown in the following example:

```
bigCat = null;
```

Passing by Value and Passing by Reference

Passing by Value and Passing by Reference

Passing by value	Copying the value of the input parameter into another variable on the stack, meaning that there are now two completely separate variables that can be changed independently.
Passing by reference	Passing a pointer to the parameter's location, meaning that only one copy of the variable exists but two things are pointing to it. Changes made to the variable inside the method are reflected in the original variable.

> *Java is a pass-by-value language.*

Instructor Demonstration

Parameter Passing

Activity: Simple Calculator

Suggested Time:

Time’s Up! Let’s Review.

Review Questions

- What does cohesion mean? Can we think of any real-world examples of cohesion?
- Why do we want classes to be cohesive?
- Give an example of something that would make a class incohesive.

The single-responsibility principle
is another name for cohesion.

It's so important that it is worth repeating.
Each class should have just one area
of responsibility.

Method Overloading

Method overloading allows us to have several methods with the same name. Why would we do this?

We could have a series of `add()` methods: one for adding ints, one for adding floats, and so on.

System.out has several `println(...)` methods; we can print Booleans, strings, ints, etc.

For this to work, each method must have a different **signature**.

Each overloaded method will have the same name but a different parameter list.

Time to Code

Method Overloading

Suggested Time:

Activity: Refactored Calculator

Suggested Time:

Recap

Recap

- Java is an OO language with some functional features.
- Properties of an object-oriented language.
- Public interface/private implementation.
- Classes vs. objects.
- Every time we create a new class, we create a new type.
- Constructors are special methods used to initialize an object upon creation.
- Classes are made up of properties and methods.

Recap

- Properties and methods marked with the static keyword are associated with the class rather than with a specific instance of the class.
- We instantiate objects using the new operator and a constructor.
- The constructor's job is to initialize the object.
- Using objects.
- Memory management.
- Java is a pass-by-value language.
- Cohesion and the single-responsibility principle are very important.
- Method overloading allows us to have methods of the same name that take different parameters, which makes code really easy to use.