TECH ELEVATOR

Java

Classes and Encapsulation

Module 1 - Week 5

Housekeeping

- You can re-submit within a module after a deadline for a max grade of 2. You must communicate this with me! This should not be a regular occurrence (TE policy is 2 late submissions per module).
- At least 2.0 must be maintained to take End of Module Assessment and move to Module 2

Schedule

Week	Topics
Week 0	Welcome, Intro to Tools
Week 1	Variables and Data Types, Logical Branching
Week 2	Loops and Arrays, Command-Line Programs, Intro to Objects
Week 3	Collections
Week 4	**Mid-Module Project**
Week 5	Classes and Encapsulation
Week 6	Inheritance, Polymorphism
Week 7	Unit Testing, Exceptions and Error Handling
Week 8	File I/O Reading and Writing
Week 9	**End-of-module project**
Week 10	**Assessment**

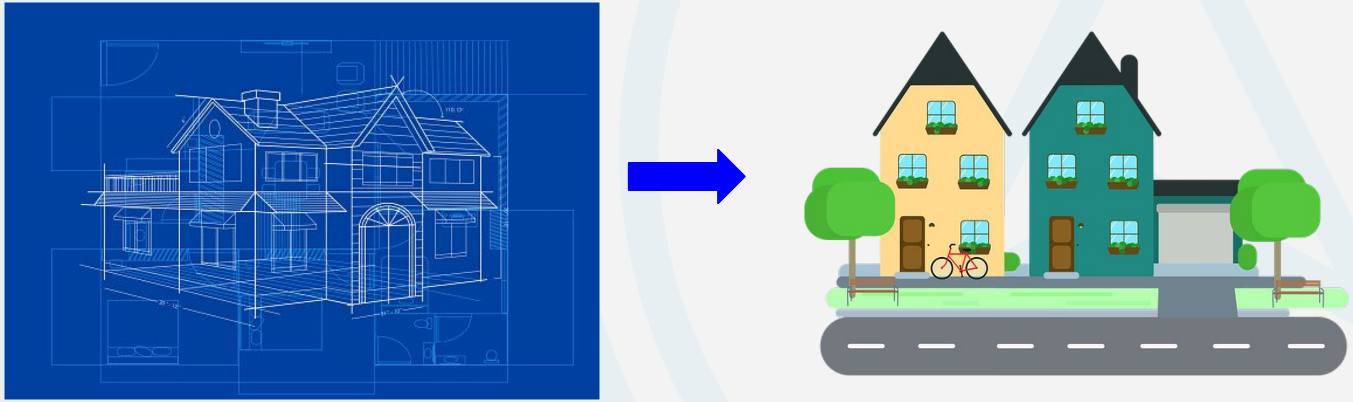
Classes & Objects

- A **`Class`** is source code; a grouping of variables and methods, used to create objects.
- An **object** is an in-memory instantiation of a Class.

The blueprint on the left was used to build the two houses on the right. The blueprint specifies that the houses will have a color but the actual color can be different for each house built. You can think of a class as the blueprint and objects as the actual houses that are built from the blueprint using their own properties, such as color.

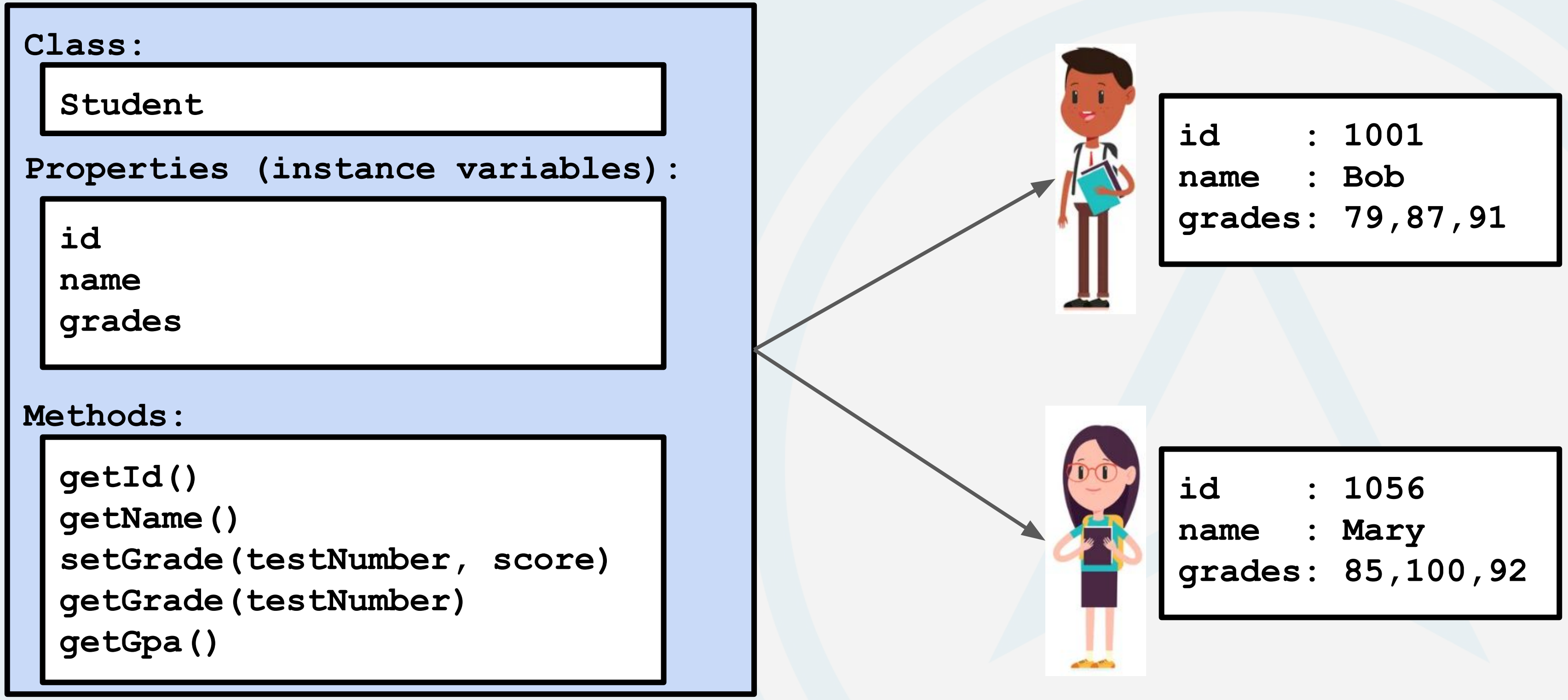
Classes & Objects

- A class is a blueprint to create an object
  - Specifies state/variables
  - Defines behavior/methods
- Class Naming
  - Use singular nouns, not verbs
  - Class must match the file name
  - Use Pascal casing
  - A ***Fully Qualified Name*** is unambiguous and includes the package and class name



Classes & Objects

- Some classes are "built in" to Java like **`String`**, **`Scanner`** and the **wrapper classes.**
- We can also define our own classes.



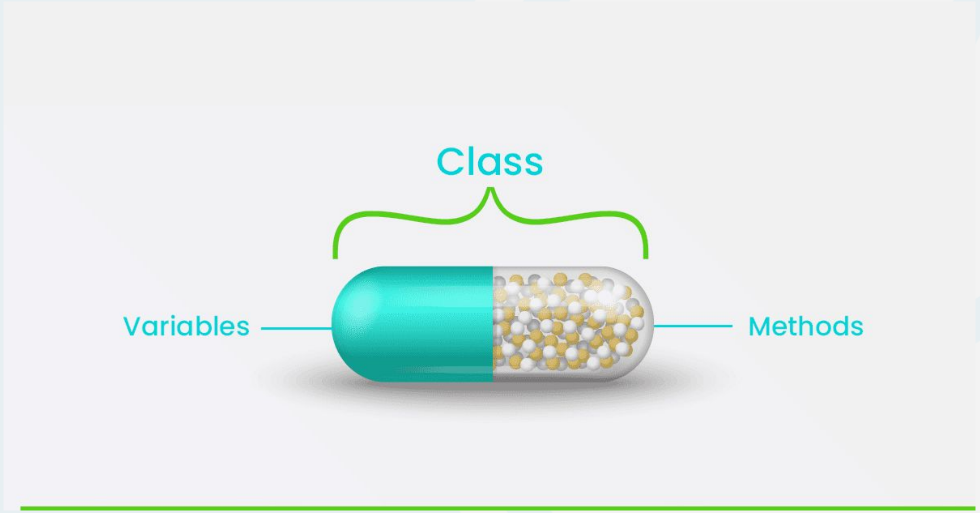
```
Class:
  Student
Properties (instance variables):
  id
  name
  grades
Methods:
  getId()
  getName()
  setGrade(testNumber, score)
  getGrade(testNumber)
  getGpa()
```

```
id     : 1001
name   : Bob
grades: 79,87,91
```

```
id     : 1056
name   : Mary
grades: 85,100,92
```

OOP Concepts

- **Encapsulation**: the concept of hiding values or state of data within a class, limiting the points of access
- **Polymorphism**: the ability for our code to take on different forms. In other words, we have the ability to treat classes generically and get specific results.
- **Inheritance**: the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes



Encapsulation

- **Encapsulation** is the concept of hiding data and controlling access to it.
- Letting other code modify data in an instance can be dangerous because it means an instance is not in control of its internal state.
- Hiding code implementation allows other classes to use a class without knowing anything about how it works
- By declaring instance variables (properties) private, we prevent other code from accessing them directly.
- We use **getters** and **setters** to provide limited access to properties by external code.
- Makes code extendable.
- Makes code maintainable.
- Encapsulation promotes **"loose coupling."**
  - A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

Describe the object oriented principle of Encapsulation - Objective 1

Vacation Expense Estimator

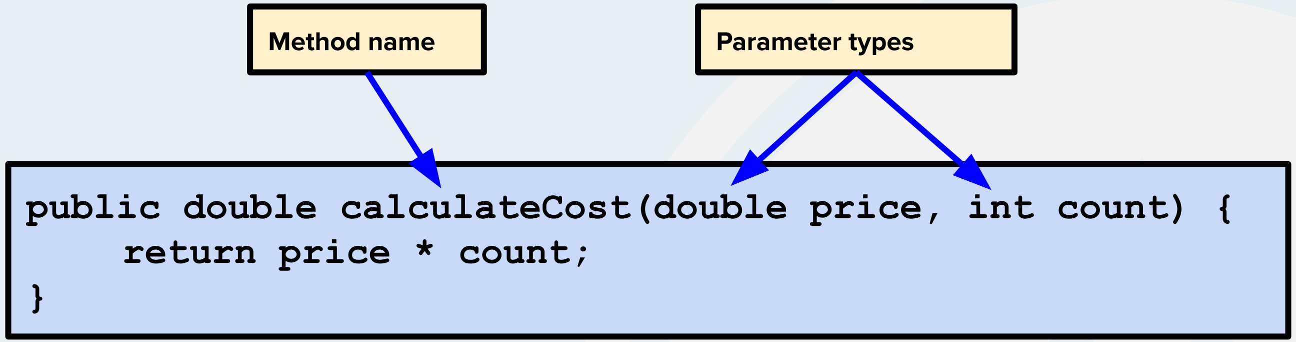
- **`com.techelevator.vee.model`** package encapsulate various aspects of a vacation
- **`com.techelevator.util`** package encapsulate general capabilities like reading and writing to files and parsing XML
- The **`Vacation`** class, for example, can include the cost of lodging when calculating the total cost of the vacation without including any knowledge or details about how to calculate the cost of lodging because the **`Lodging`** class encapsulates those details.

Objectives

- Describe the object oriented principle of Encapsulation
- **Identify the key elements of a class declaration (name, member variables, constructors, methods)**
- Use the private and public access modifiers appropriately to hide or expose elements of a class

Methods Signature

- The two components of a method declaration that make up signature are the method's name and it's parameter types.



```
public double calculateCost(double price, int count) {
    return price * count;
}
```

- The signature of the above method is…

```
public double calculateCost(double)
```

- No two methods in the same class can have the same signature.

Getters & Setters

- Getters and Setters (aka accessors) allow public access to a private member variable while still allowing the class to have full control of the variable. .
- Should be the only way to access member variables from outside the class
- Accessors should always begin with the "get" or "set" prefix, except for getter methods that return a boolean, which usually begins with the prefix "is".

Member variable has private access

The private member variable is accessed through public methods

```
public class Student {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String studentName) {
        name = studentName;
    }
}
```

Constructors

- Every class has a constructor which is called when an object is being created.
- Constructors are defined similarly to a method but have
- The same name as the class
- No return types

```
public Student() {

}
```

- To create an object, code must call at least one constructor.
- Java provides a built-in no-argument constructor by default so that each class is not required to provide one to allow basic object creation.
- A class may declare alternate constructors with arguments.
- However, as soon as one or more constructors is explicitly declared in the class, the default constructor is no longer available so if the class needs to allow creation of objects with no arguments, the class must explicitly declare a no-argument constructor to replace the default constructor which is no longer available.

Here, the **`Student`** class does not explicitly declare a constructor.

```
public class Student {
    // some code…
    // but no defined constructor
}
```

By default, Java provides a no-argument constructor so even though no constructor is explicitly declared for the Student class, we can still create a Student object using a constructor with no arguments.

```
Student student1 = new Student();
```

this

- This this keyword refers to the member variable specific to the instance of an object where the code is run.
- We can use this to resolve name collisions in constructors and setters.

```
public class Student {
    private final String name;
    private final int id;

    public Student(String name, int studentId) {
        this.name = name;
        id = studentId;
    }
    // some more code…
}
```

```
Student student1 = new Student("Walter", 99);
Student student2 = new Student("Jesse", 100);
```

The constructor argument **`name`** has the same name as one of the instance variables.

We use the **`this`** keyword to differentiate between the instance variable and the constructor argument.

Because the argument being assigned to **`id`** has a different name, the **`this`** keyword is not required.

Derived Properties

A derived property is a getter that, instead of returning a member variable, returns a calculated value. If we have grades, we can derive an average value from them.

```java
public class Student {
      private final Map<Integer, Double> grades = new HashMap<>();

    // other code here…

    public double getAverage() {
        double total = 0.0;
        for (Map.Entry<Integer, Double> gradeEntry : grades.entrySet()) {
            total += gradeEntry.getValue();
        }
        return total / grades.size();
    }
}
```

This method exposes a derived property

Overloading

Overloaded methods are methods with the same name and return type, and a different set of parameters. Java uses the correct overload based on the parameters sent to it.

```
public class Student {
  // some more code…

  public void addGrade(int testNumber, double grade) {
      grades.put(testNumber, grade);
  }

  public void addGrade(int testNumber, double grade, double extraCredit){
      grades.put(testNumber, grade + extraCredit);
  }
}
```

This overloads the **addGrade** method above.

static

Static members belong to the class. Instance members belong to an instance of the class. Static methods can be invoked without creating an instance of the class.   Static variables and methods cannot be accessed with the this keyword, since they are not part of the object.

```
private static int x = 10;

int y = x + 5;
int y = this.x + 5;  // the keyword this cannot be used with static
```

Since static variables are shared by all instances of a class and not the individual object. Changes to the value from one object can be seen from all objects of that type.

If we define a method static, our class does not need to be instantiated to use it. Instead it can be accessed from the `Class` itself, instead of the object. Static methods can only access other static methods or variables.

This seems easier, why not make everything static?

Identify the key elements of a class declaration - Objective 2

Common difficulties

- knowing when to use **`this`**
- creating unnecessary variables for derived properties
- understanding **`static`** variables and methods

Constructors

Here, the Student class declares a constructor which takes a String and an int as arguments.

```
public class Student {
    private final String name;
    private final int id;

    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }
    // some more code…
}
```

We can now create a Student object using this new constructor.

However, because we have declared our own constructor **the default no-argument constructor is no longer available so this is NOT valid.**

```
// Valid
Student student1 = new Student("Walter", 99);

// No longer valid
Student student2 = new Student();
```

UML Diagrams - Teams to breakout rooms

1. Navigate to draw.io web app.

2. Start with a blank template.

3. From the UML dropdown, create a Class 2 component for each class in the **src/main/java/com/techelevator/vee/model** directory.

4. Add connections between classes using lines/arrows.

5. When we are all back from the breakout rooms, we can discuss any questions or thoughts you have.

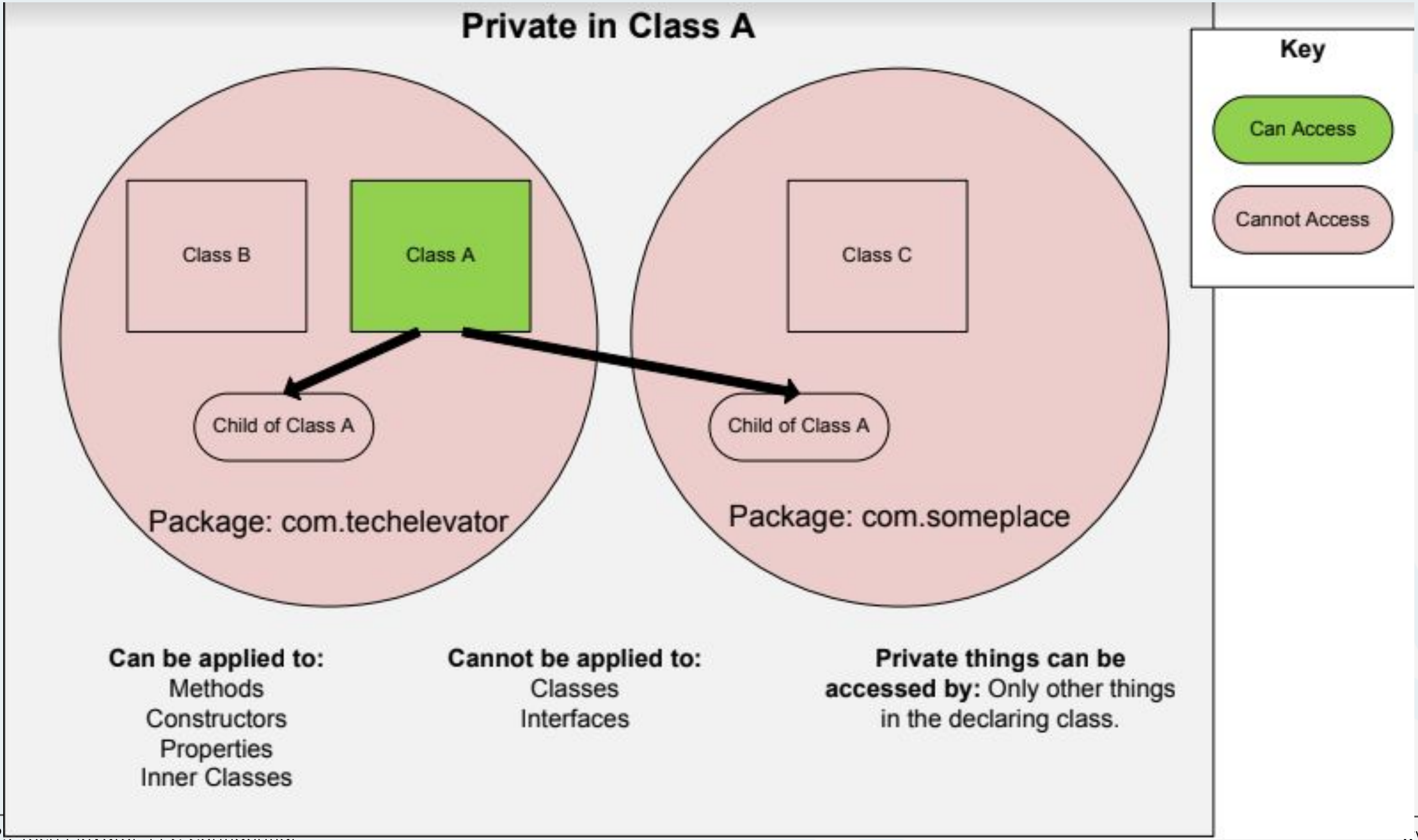
Objectives

- Describe the object oriented principle of Encapsulation
- Identify the key elements of a class declaration (name, member variables, constructors, methods)
- Class design should include how others will use your object, the methods that allow that use should be public. All other methods and variables should be private, until needed in the hierarchy or publically.
- **Use the private and public access modifiers appropriately to hide or expose elements of a class**

Protected and Default Access Modifiers

- **Private** - accessible on in the class
  - can be applied to methods and member variables
- **Protected** - accessible in the class and in any subclasses in the inheritance tree.
  - can be applied to methods and member variables
  - In Java, Protected it also available to any class in the same package, but this use is discouraged.
- **Default (no access modifier)** - accessible to any class or subclass in the same package.
  - can be applied to methods and member variables
- **Public** - accessible everywhere
  - can be applied to methods, member variables, classes, and interfaces)

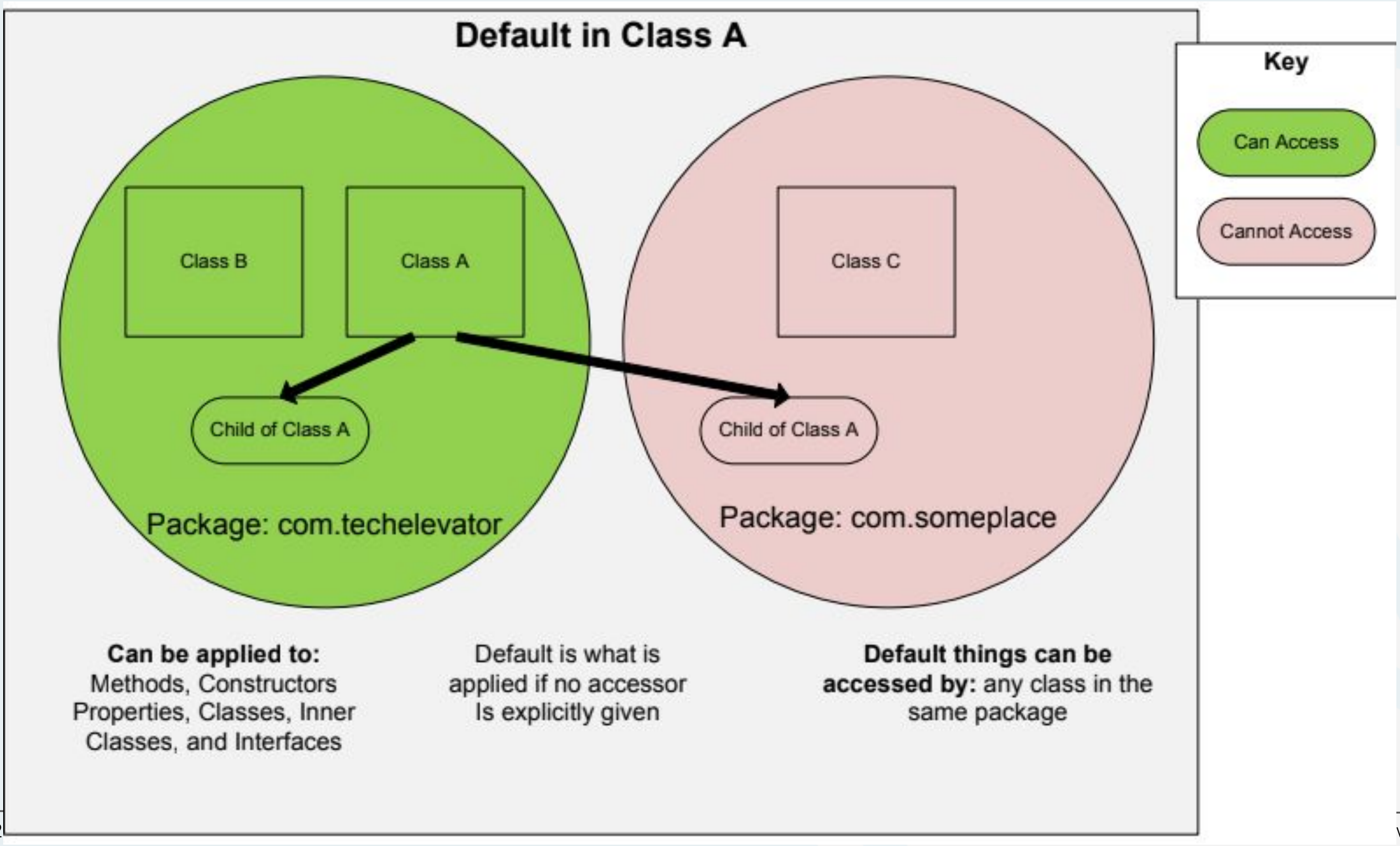
Private in Class A



Can be applied to:
Methods
Constructors
Properties
Inner Classes

Cannot be applied to:
Classes
Interfaces

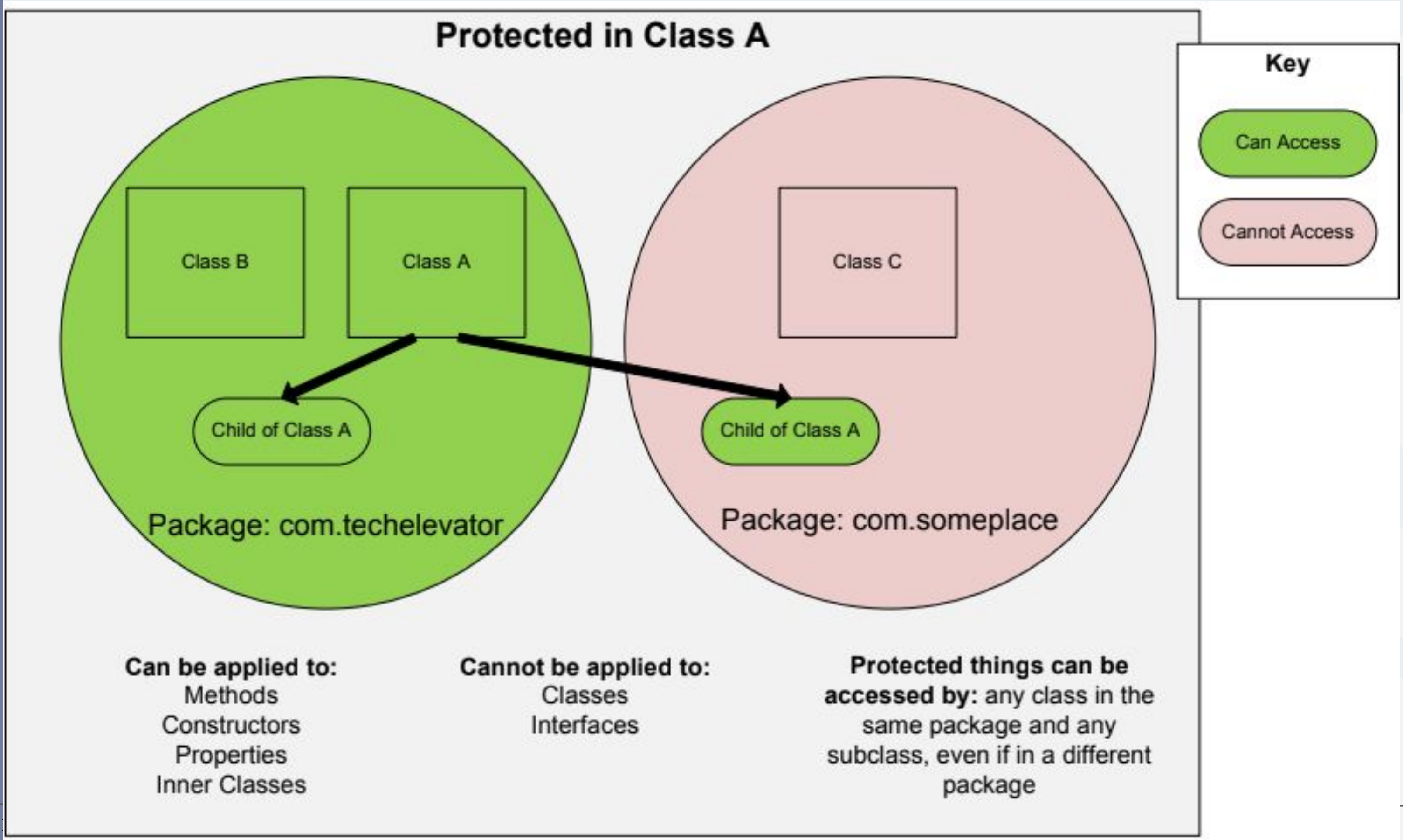
Private things can be accessed by: Only other things in the declaring class.



Can be applied to: Methods, Constructors Properties, Classes, Inner Classes, and Interfaces

Default is what is applied if no accessor Is explicitly given

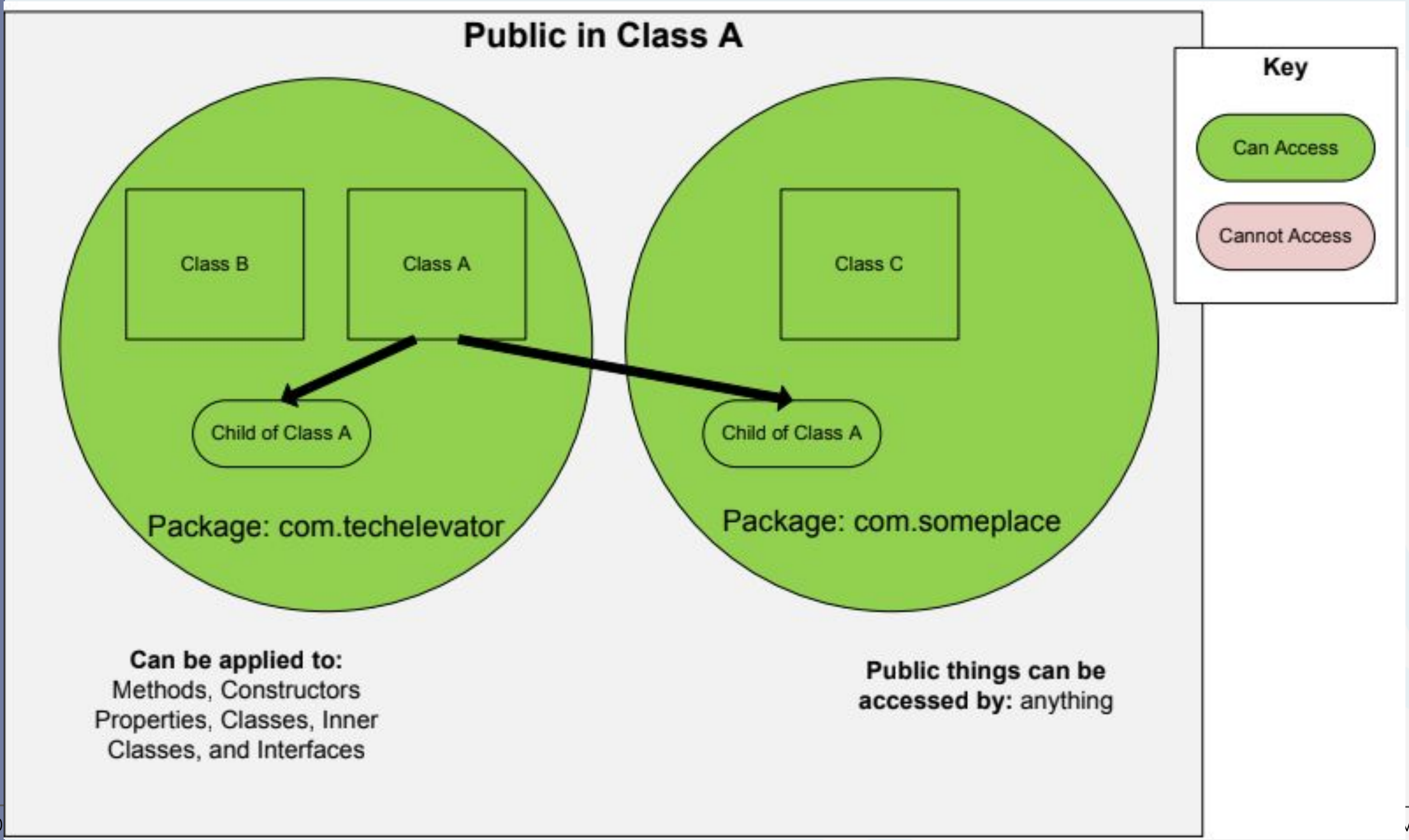
Default things can be accessed by: any class in the same package



Can be applied to:
Methods
Constructors
Properties
Inner Classes

Cannot be applied to:
Classes
Interfaces

Protected things can be accessed by: any class in the same package and any subclass, even if in a different package



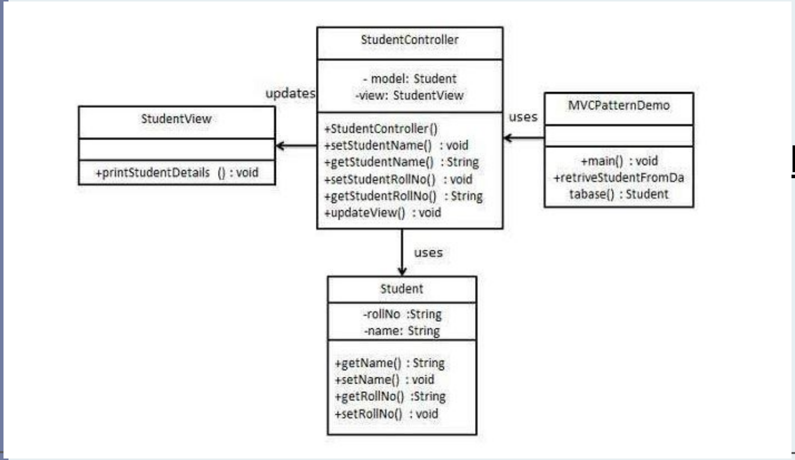
Can be applied to:
Methods, Constructors
Properties, Classes, Inner
Classes, and Interfaces

Public things can be accessed by: anything

Access	**Visibility**	**Reason to use it**
public	Everyone	for “set in stone” methods that you want other programmers to rely on to use your object. These create the behaviors of the object, but changing their method signatures may break other code that is using your object.
protected	Subclasses	for building connections between inherited classes. It lets you have methods in a superclass that are accessible to the subclasses, but does not allow access outside the hierarchy.
default	Package	for building cohesion between related classes in the same package. should generally be avoided.
private	Class	for unstable, worker methods that may change and are only for use inside the class itself.

Design Patterns

- Represent best practices used by experienced object-oriented software developers.
- Solutions to general problems that software developers faced during software development..



https://www.tutorialspoint.com/design_pattern/index.htm

final

Sometimes we have constant values, or values that can not be changed, that we want to use in our code without duplicating the value all over the place. Many languages have the concept of constant and global constant variables, however, Java does not, but we can mimic one using the final keyword.

The final keyword allows the variable to be assigned once, but after set, it cannot be reassigned.

```
private final int x = 10;
x = 15;  // not allowed, since x is final and has a value

private final int y;
y = 20;   //  allowed, y was declared as final, but not yet assigned
y = 30;   // not allowed, since y is final and has a value.
```

Instance Variables

Instance variable represent the properties of a class.

- Each instance of a class will have its own values for instance variables that represent its internal state.
- Instance variables are declared with access modifiers.
    - public - can be accessed by any other object.
    - private - can only be accessed by the current instance of a class.

```
public class Student {
    private final String name;
    private final Map<Integer, Double> grades = new HashMap<>();

    // more code here …
}
```

Variables Summary

- **Member:** Variable declared on a class.
    - **Instance**: **Non-static** fields are also known as instance variables because their values are unique to each instance of a class.
    - **Class**: A class variable is any field declared with the **static** modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

```
public class Circle {
    public static final double PI = 3.14159265358979323846;

    // more code here …
}
```

- **Local**: Declared in a method. Only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameter**: Variable passed into a method.   The value is set by the arguments passed when the method is called.

Use the private and public access modifiers expose elements - Objective 3

Common difficulties

- having a bad habit of making everything **`public`**
- unintentionally using the default access modifier by forgetting to specify one
- mistakenly considering **`static`** or **`final`** to be access modifiers