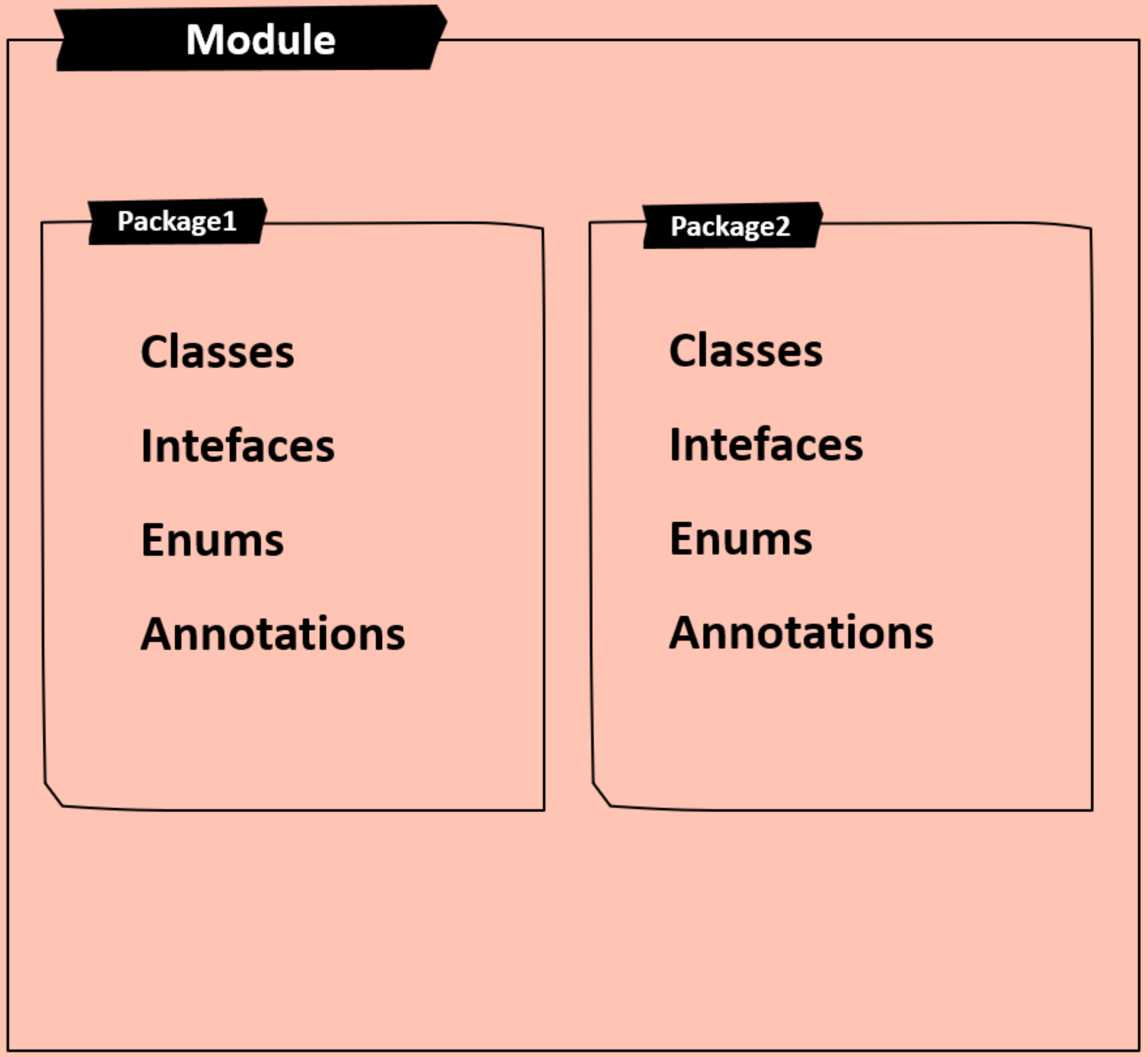
Structuring of code in Java projects

eazy bytes



Modules were introduced with JDK 9. Prior to JDK 9, packages and types existed, but modules were not part of the structure. Modules serve as an optional means of organizing code, complementing the existing packages.

A Java project is comprised of one or more modules, each housing zero or more packages. Within a package, there exist one or more types, with "type" being a general term encompassing user-defined data types. The four main user-defined data types are classes, interfaces, enums, and annotations. More details about interfaces, enums & annotations will be shared in coming sections.

Understanding packages in Java is mandatory since every real projects group their classes, interfaces etc. into various packages for various benifits. That’s why we are going to focus on package for next few lectures.

Since modules are optional and introduced as part of JDK9, till today many real projects are not using modules. We will talk about modules in the coming sections of the course, once we discuss about all the basics of Java

packages in Java

eazy bytes

In Java, packages are used to organize classes into namespaces. A package is a collection of related classes, interfaces, annotations and enums grouped together in a single unit. Advantages of packages are mentioned below,

1. **Organization of the code**

   Packages help to organize large codebases into logical units, making it easier to manage and maintain code.

2. **Encapsulation**

   Packages provide a way to encapsulate classes and interfaces, hiding implementation details from other parts of the code.

3. **Access control**

   Packages can be used to control access to classes and interfaces, using access modifiers like public, private, protected, and default.

4. **Namespacing**

   Packages provide a way to avoid naming conflicts between classes and interfaces, by giving them unique names within a package.

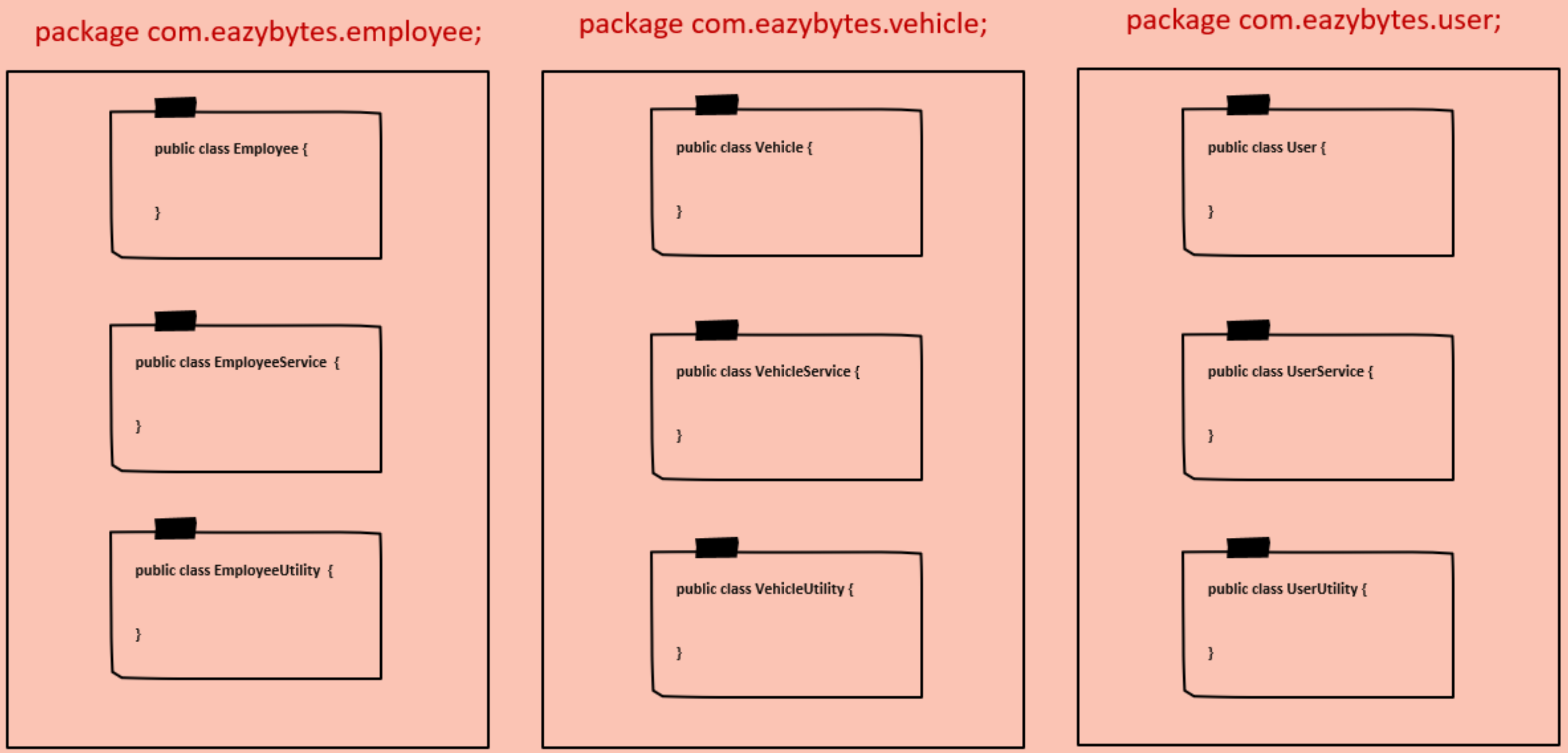
If you do not use a package statement, your type ends up in an unnamed or default package. Generally speaking, an unnamed or default package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes, interfaces, annotations and enums belong in named packages.

Creating a Package

To attach a class to a package, you can use the package keyword followed by the package name, at the top of your source code file. For example:

```java
package com.example.myapp;

public class MyClass {
    //class code goes here
}
```

The package statement must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.



package com.eazybytes.employee;

```java
public class Employee {
}

public class EmployeeService {
}

public class EmployeeUtility {
}
```

package com.eazybytes.vehicle;

```java
public class Vehicle {
}

public class VehicleService {
}

public class VehicleUtility {
}
```

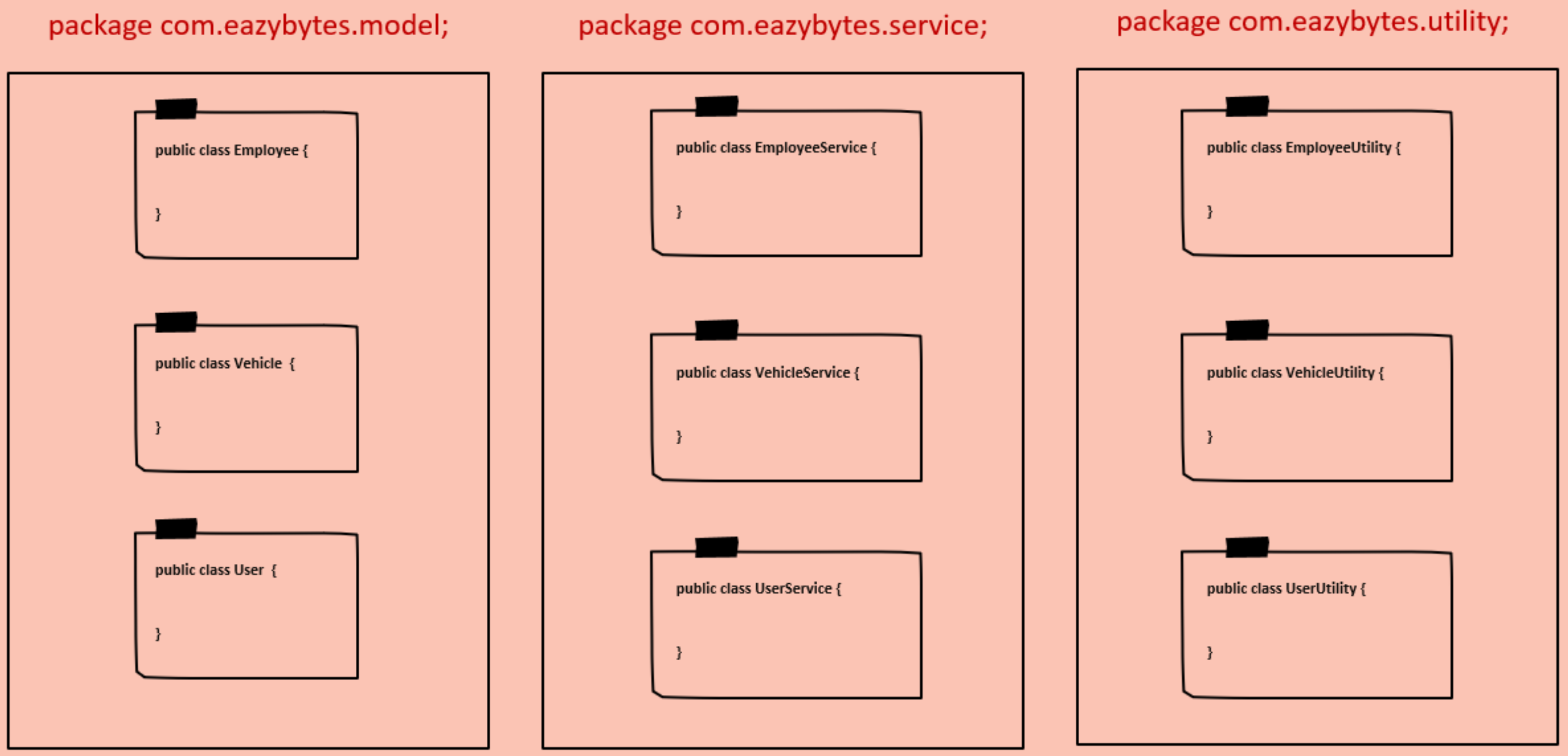
package com.eazybytes.user;

```java
public class User {
}

public class UserService {
}

public class UserUtility {
}
```

As demonstrated above, classes can be organized into logical groups based on their functionality or purpose. Such related classes can be assigned a common package name in Java.

Creating a Package

eazy bytes


package com.eazybytes.model;
public class Employee { }
public class Vehicle { }
public class User { }
package com.eazybytes.service;
public class EmployeeService { }
public class VehicleService { }
public class UserService { }
package com.eazybytes.utility;
public class EmployeeUtility { }
public class VehicleUtility { }
public class UserUtility { }

As demonstrated above, classes can be organized into logical groups based on their layers based on functionality and responsibility.

In Java, there are various architectural patterns that involve dividing the application into different layers based on functionality and responsibility. Some common layers used in Java applications are:

Model layer: The model layer is responsible for representing the data of an application.

Service layer: The service layer is responsible for implementing business logic and processing data received from the model layer. It typically contains classes that implement service interfaces and perform data validation, transformation, and other business logic.

Utility layer: The utility layer provides common functionality that can be reused across the application. It typically contains helper classes, constants, and utility methods.

Controller/View layer: The controller/view layer is responsible for handling user input and presenting data to the user. It typically contains classes that handle HTTP requests, map URLs to service methods, and render views for display.

Presentation layer: The presentation layer is responsible for presenting data to the user in a format that can be easily understood. It typically contains classes that implement user interfaces, such as web pages or mobile app screens.

These layers help to separate concerns and maintain code modularity, making it easier to maintain and extend the application over time.

Naming a Package

eazy bytes

When Java programmers create classes and interfaces, it is common for them to use identical names, which can cause confusion if they have different meanings. Nevertheless, the Java compiler permits two classes with the same name as long as they belong to different packages. However, this approach can lead to problems if two independent programmers choose the same package & class name for their work. To prevent this problem, there is a convention that programmers follow.

The rules to follow while naming a package are,

1. Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
2. Companies use their reversed Internet domain name to begin their package names—for example, com.oracle.mypackage for a package named mypackage created by a programmer at oracle.com
3. Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region, department or the project name after the company name (for example, com.oracle.sales.mypackage).
4. Packages in the Java language itself begin with java. , jakarta. or javax.
5. In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name. In this event, the suggested convention is to add an underscore. For example below are the domain names and their corresponding package names:

```
for-ever.example.org  -> org.example.for_ever
123.example.com -> com.example._123
```

Using package members with import statement

The elements that make up a package are referred to as its package members. To utilize a public package member from outside the package, you need to take one of the following actions:

1. Reference the member by its complete, fully qualified name.
2. Import the package member.
3. Import the entire package that the member belongs to.

1 Using fully qualified name

if you are trying to create a object of Vehicle class from a different package, you must use the member's fully qualified name like shown below,

```
com.eazybytes.vehicle.Vehicle veh = new
com.eazybytes.vehicle.Vehicle();
```

2 importing the package member

To import a specific package member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one. Here's how you would import the Vehicle class from it's package

```
import com.eazybytes.vehicle.Vehicle;
```

Now you can refer to the Vehicle class by its simple name.

```
Vehicle veh = new Vehicle();
```

3 importing the entire package

If you want to import all the types that are present in a particular package, you can make use of the import statement along with the asterisk (*) wildcard character.

```
import com.eazybytes.vehicle.*;
```

Now you can refer to any class or interface in the above package by its simple name.

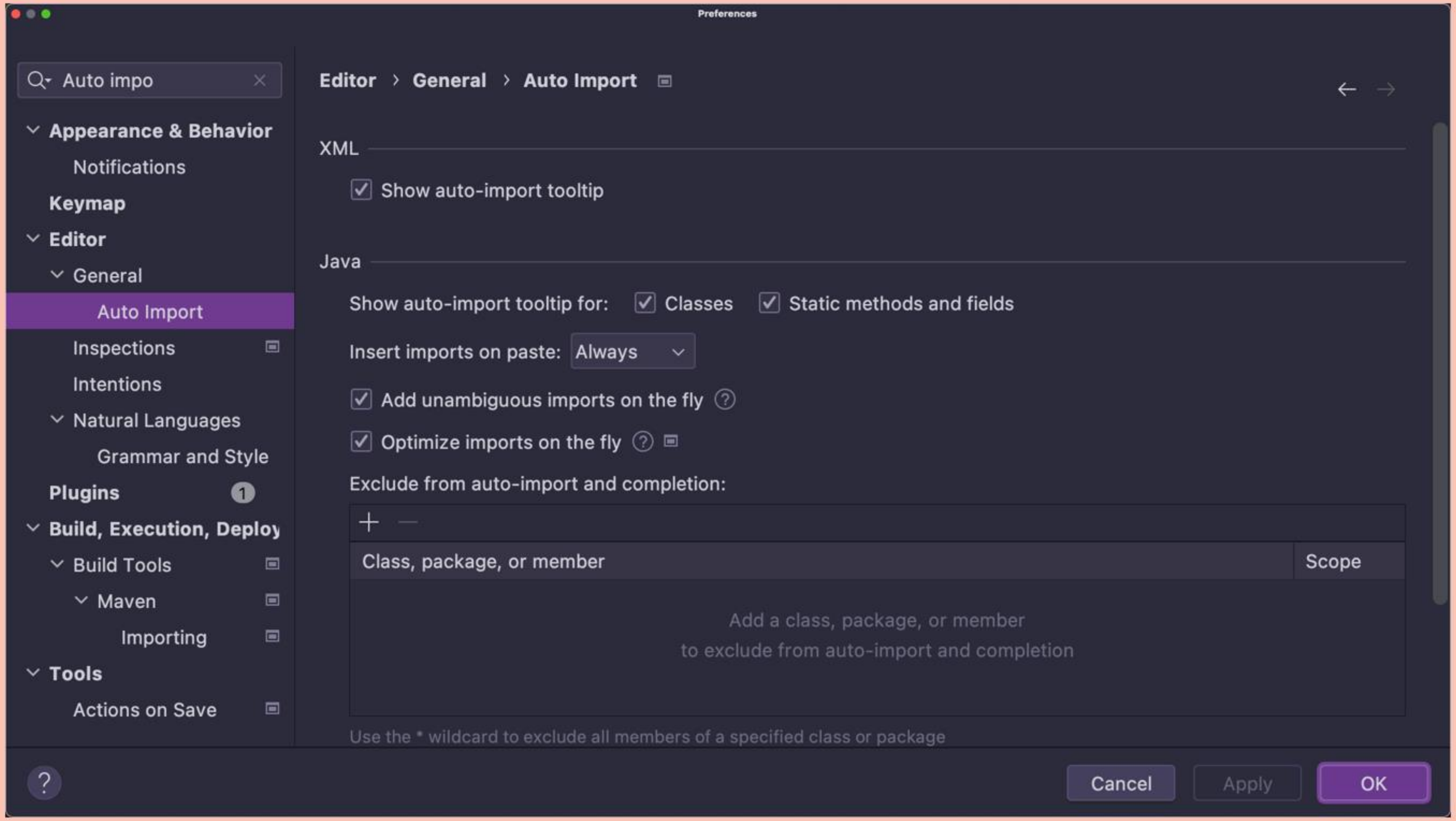
```
Vehicle veh = new Vehicle();
```

The asterisk in the import statement can be used only to specify all the classes within a package & cannot be used to match a subset of the classes in a package. We generally import only a single package member or an entire package.

IntelliJ IDEA Auto Import management

eazy bytes

You can make your IntelliJ IDEA to take care of import statements automatically based on the code that you add or remove. For the same, navigate to the Preferences & look for Auto Import. Post that please enable all the check boxes like you can see in the below screenshot,



Editor › General › Auto Import

XML

- [✓] Show auto-import tooltip

Java

Show auto-import tooltip for: [✓] Classes [✓] Static methods and fields

Insert imports on paste: Always

- [✓] Add unambiguous imports on the fly
- [✓] Optimize imports on the fly

Exclude from auto-import and completion:

Class, package, or member	Scope

Add a class, package, or member
to exclude from auto-import and completion

Use the * wildcard to exclude all members of a specified class or package

java.lang package

In Java, the java.lang package is a special package that is automatically imported into any Java program. This means that any class or interface defined in the java.lang package can be used directly in a Java program without the need for an explicit import statement.

The java.lang package contains many fundamental classes and interfaces that are essential to the core functionality of the Java language. Some of the most commonly used classes in the java.lang package include:

String: The String class represents a sequence of characters, and is used extensively in Java programs for manipulating and displaying text.

Object: The Object class is the root of the Java class hierarchy, and provides basic methods for all objects in Java.

Integer, Double, Boolean, and other primitive wrapper classes: These classes provide a way to represent primitive data types (such as int, double, and boolean) as objects, which can be useful in certain situations, such as when working with collections or using Java's reflection API.

Throwable and its subclasses (Exception and Error): These classes are used to represent exceptions and errors in Java programs, and are essential for handling errors and other unexpected conditions.

Math: The Math class provides a set of static methods for performing common mathematical operations, such as trigonometric functions, logarithms, and random number generation.

Because the java.lang package is automatically imported into all Java programs, you can use these classes and interfaces directly without needing to include an explicit import statement. However, it's important to note that other packages and classes must be imported explicitly using an import statement before they can be used in a Java program.

The static import statement

In scenarios where frequent access to static final fields (constants) and static methods from one or two classes is necessary, continually prefixing the class names can lead to code clutter. The static import statement provides a solution by allowing you to import the specific constants and static methods you need, eliminating the need to repeatedly prefix their class names.

The java.lang.Math class provides a collection of mathematical functions and constants that are commonly used in programming. These functions and constants can be accessed without explicitly specifying the Math class by using the static import statement.

```java
import static java.lang.Math.PI;
double radius = 5.0;
double circumference = 2 * PI * radius;
System.out.println("Circumference: " + circumference);
```

In this example, the cos function is imported using the static import statement. This allows us to use cos without prefixing it with Math.

```java
import static java.lang.Math.cos;

double angle = 30.0;
double cosine = cos(angle);
System.out.println("Cosine: " + cosine);
```

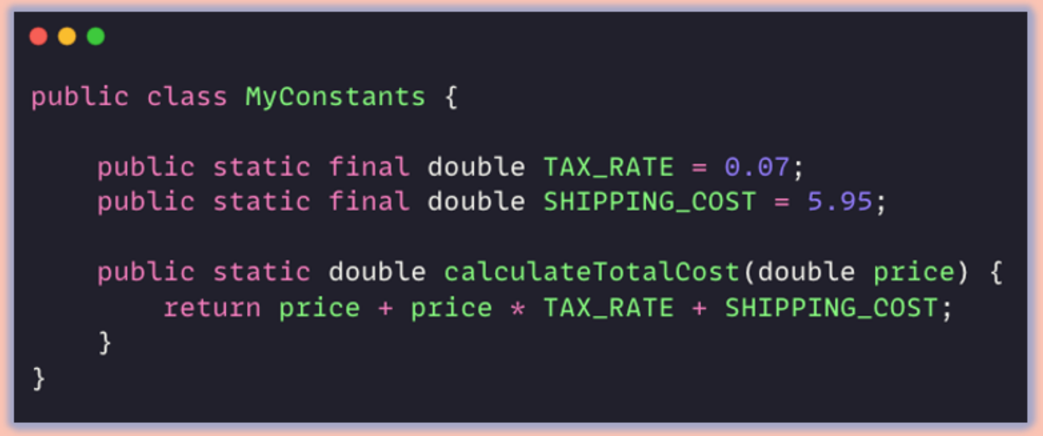
The static members of Math can be imported either individually or as a group like shown below.

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
```

```java
import static java.lang.Math.*;
```

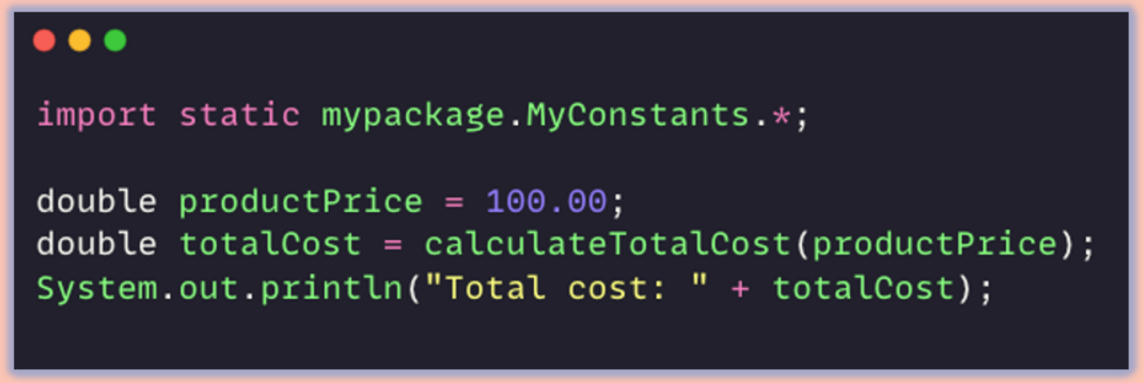
The static import statement

You can also use the static import statement with custom classes that contain constants and static methods that you use frequently. For example, consider the following class:



```java
public class MyConstants {

    public static final double TAX_RATE = 0.07;
    public static final double SHIPPING_COST = 5.95;

    public static double calculateTotalCost(double price) {
        return price + price * TAX_RATE + SHIPPING_COST;
    }
}
```

In this example, both the TAX_RATE constant and the calculateTotalCost method are used without prefixing them with MyConstants. This makes the code more concise and easier to read.



```java
import static mypackage.MyConstants.*;

double productPrice = 100.00;
double totalCost = calculateTotalCost(productPrice);
System.out.println("Total cost: " + totalCost);
```

Other Important points about packages & imports

eazy bytes

Doesn't allow recursive imports

Although it may seem intuitive for a wildcard import to encompass both the classes of the specified package and those of its subpackages, Java doesn't support recursive imports. If you require classes from both java.util and java.util.regex, you must use separate import lines for each package.

Doesn't support nesting structure

In Java, packages do not have a nesting structure. For instance, the packages java.util and java.util.regex are unrelated and function as independent collections of classes.

Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the graphics package defined a class named Rectangle. The java.awt package also contains a Rectangle class. If both graphics and java.awt have been imported, the following is ambiguous.

```
Rectangle rect;
```

In such a situation, you have to use the member's fully qualified name to indicate exactly which Rectangle class you want. For example,

```
graphics.Rectangle rect;
```

Don't overuse static imports

Exercise caution when employing static imports in Java and limit their use. Excessive static imports can lead to code that is challenging to comprehend and maintain, as it becomes unclear which class defines a specific static object. When used judiciously, static imports enhance code readability by eliminating the need for repetitive class names.

access modifiers in Java

In Java, access modifiers are keywords used to specify the visibility of classes, methods, and variables. There are four access modifiers in Java.

1. **public** : A public class, method or variable can be accessed from anywhere, inside or outside the class, in any other class or package.
2. **private:** A private class, method or variable can only be accessed within the same class where it is declared. It cannot be accessed from any other class or package.
3. **protected:** A protected class, method, or variable can only be accessed by classes in package and subclasses inside or outside the package.
4. **default:** If no access modifier is specified, then the class, method or variable is given package-private access. This means that the class, method or variable can only be accessed within the same package, but not from outside the package.

It is good practice to use access modifiers to control access to your classes, methods, and variables. This can help to prevent unintended modification or access to your code, which can result in bugs and security vulnerabilities.

access modifiers in Java

eazy bytes

Here is an example of access modifiers in Java:

```java
public class MyClass {
    public int publicVar;
    private int privateVar;
    protected int protectedVar;
    int defaultVar; // package-private access by default

    public void publicMethod() {
        // method body
    }

    private void privateMethod() {
        // method body
    }

    protected void protectedMethod() {
        // method body
    }

    void defaultMethod() {
        // method body
    }
}
```

In this example, the publicVar and publicMethod() can be accessed from anywhere, the privateVar and privateMethod() can only be accessed within the same class, the protectedVar and protectedMethod() can be accessed within the same package or in any subclass of MyClass, and defaultVar and defaultMethod() can be accessed within the same package only.

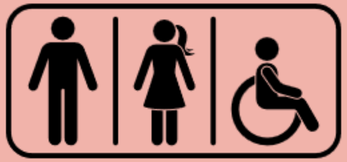
access modifiers in Java

eazy bytes

Here are some analogies to understand access modifiers in terms of public toilets and private toilets:

public

1



public members are accessible from anywhere, just like public toilets that are accessible to everyone.

private

2

private members are only accessible within the same class, just like a toilet inside a house that is only accessible to the residents.

default

3

default members are accessible within the same package, just like a toilet in a office building that is only accessible to the people who work in that building.

protected

4

protected members are accessible within the same package and subclasses, just like a toilet in a shared apartment that is accessible to the residents and their guests.

access modifiers in Java

The following table below displays the access levels for different modifiers in Java,

	public	private	default	protected
Same class	Yes	Yes	Yes	Yes
subclass in same package	Yes	No	Yes	Yes
non-subclass in same package	Yes	No	Yes	Yes
subclass in different package	Yes	No	No	Yes
non-subclass in different package	Yes	No	No	No

The access modifiers build on each other where you start with no access to the outside world (private) and add to it: first, package (default); then, subclasses (protected); and then the world (public).

POJO (Plain Old Java Object) classes

In Java, POJO (Plain Old Java Object) classes are used to represent entities or data objects. These classes contain properties (variables) that hold data, but to access or modify this data from other classes, we need to use getter and setter methods.

A class method that retrieves the value of a private field and returns it is called a getter. On the other hand, a setter is a class method that assigns a value to a private field.

DTO (Data Transfer Object) and JavaBean are alternative terms for POJO.

eazy bytes

Here are some reasons why we need getter and setter methods in POJO classes:

1. **Encapsulation** : Encapsulation is a fundamental concept of object-oriented programming that helps to protect the internal state of an object from being accessed or modified by other classes directly. Getter and setter methods provide a way to access and modify the internal state of an object in a controlled manner, without exposing its implementation details.

2. **Data validation:** Getter and setter methods can be used to validate the data being set or retrieved from an object. For example, we can use a setter method to validate that a particular property value is within a certain range or meets some other criteria.

3. **Flexibility:** Getter and setter methods provide flexibility in changing the implementation of a class without affecting the clients that use it. For example, we can add additional logic to a setter method, such as logging or triggering some other action, without changing the public interface of the class.

4. **Access control:** Getter and setter methods allow us to control the access to the properties of an object. We can use the access modifiers such as private or protected to restrict the access to certain properties, and provide getter and setter methods with appropriate access modifiers to provide controlled access to those properties.

In summary, getter and setter methods provide a way to encapsulate the data of a POJO class and provide controlled access to it. They also provide a way to validate the data, add additional logic, and control the access to the properties of the object.

Java supports Object-oriented programming (OOP)

eazy bytes

In a scenario where you are going to work on a bank project or application, you may write Java logic that revolve around classes and objects like Customer, Account, Loan, Card, Transactions etc.

Where as the developer who works on a health care project or application, may write Java logic that revolve around classes and objects like Patient, Doctor, Treatment, Medicines, Reports etc.

Some of the key concepts of Java OOPs are:

Class

Object

Interface

Inheritance

Encapsulation

Polymorphism

Abstraction

Object-oriented languages prioritize data over imperative commands, in contrast to non-OOPs languages that emphasize providing computers with "Do this/Do that" commands. Although object-oriented programs still issue commands to the computer, they first organize the data with the help of class & objects before executing any commands. Due to this, OOPs has many advanatges like Reusability, Modularity, Easy maintenance, Encapsulation, Inheritance & Polymorphism.

Encapsulation

eazy bytes

In Object Oriented Programming, encapsulation generally refers to two concepts.

Firstly, it involves grouping attributes(fields) and behaviors(methods) together within a single object (Class). This is very similar to capsules which contain the active ingredient, along with other substances that help to stabilize and preserve the drug.

Secondly, it involves the practice of hiding certain fields and methods from public access using access modifiers like private, protected etc. This is very similar to a brief case where we hide certain data inside it in a secured manner.