SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution
Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELIGENCE AND MACHINE LEARNING

FUNDAMENTALS OF JAVA

III YEAR - VI SEM

UNIT 3 –Polymorphism and Interface

Topic 3 – Polymorphism

Java Polymorphism

What Is Polymorphism?

Polymorphism is the concept of one entity providing multiple implementations or behaviours. (Something like an Avatar!)

Java Polymorphism

Java And Polymorphism

Java provides multiple forms of polymorphism. It allows you to define the same method name to do multiple different things. It also allows child classes to redefine/ override parents' behaviours for the same method.

Java Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form.

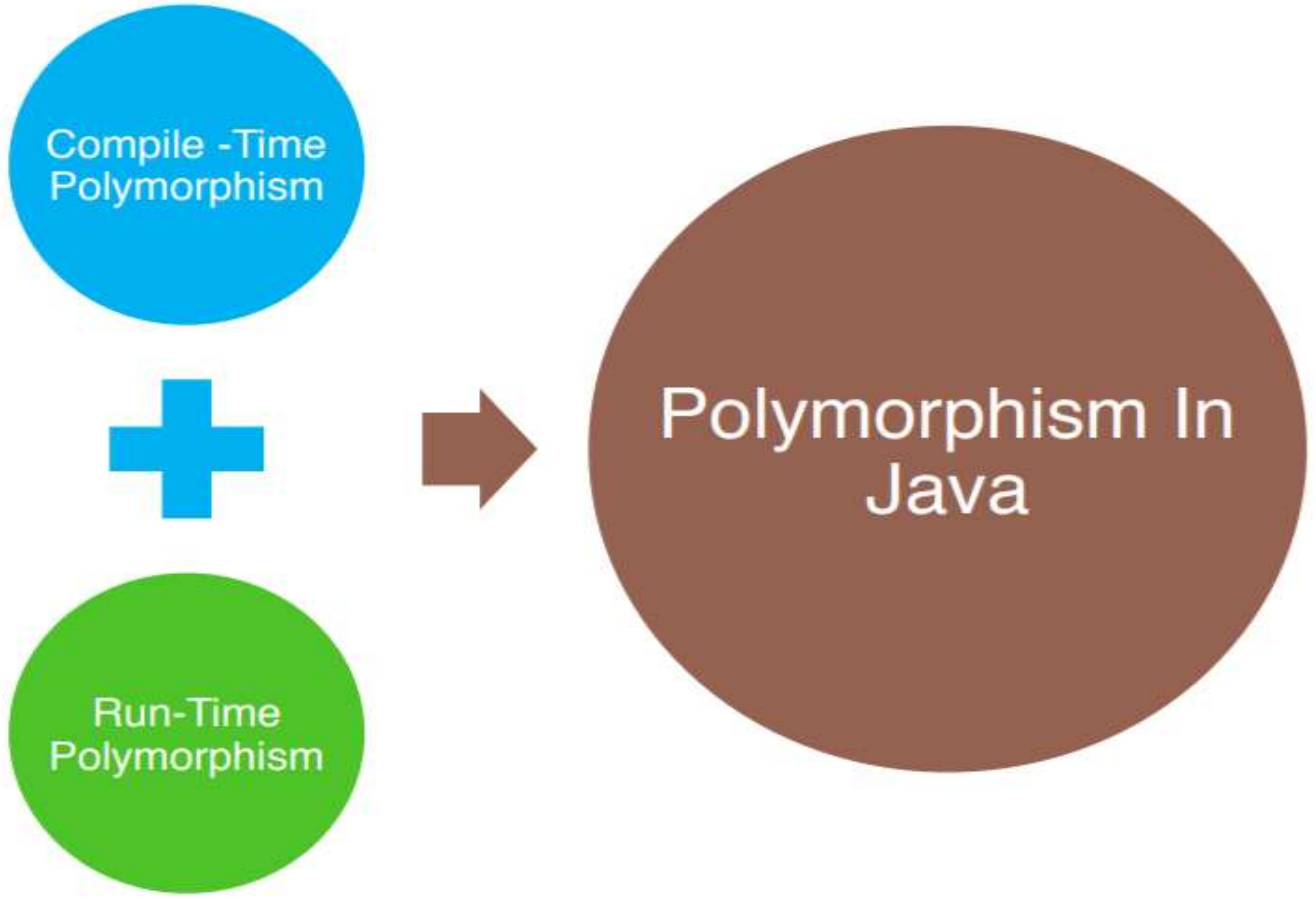
That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

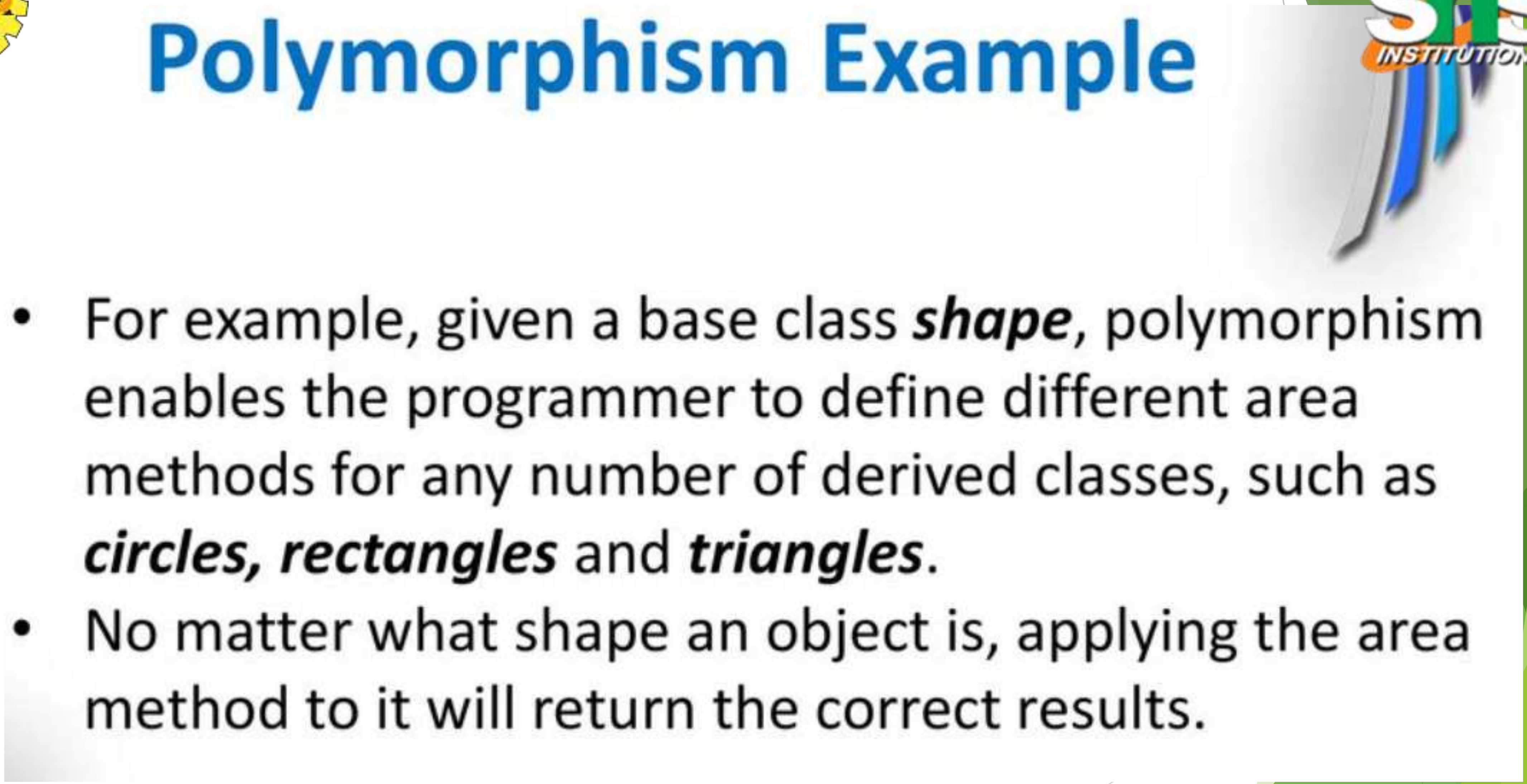
Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

Java Polymorphism

Java Provides 2 Types Of Polymorphi



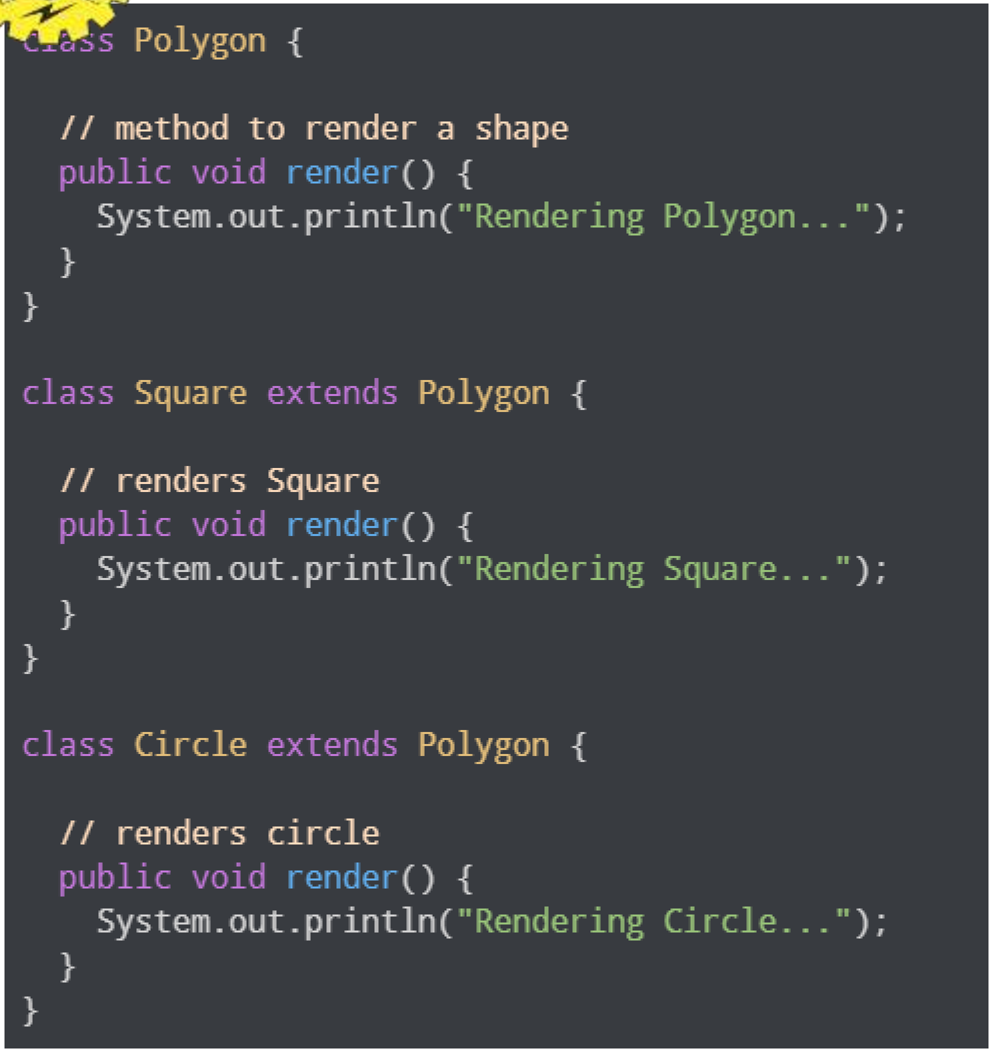
Java Polymorphism



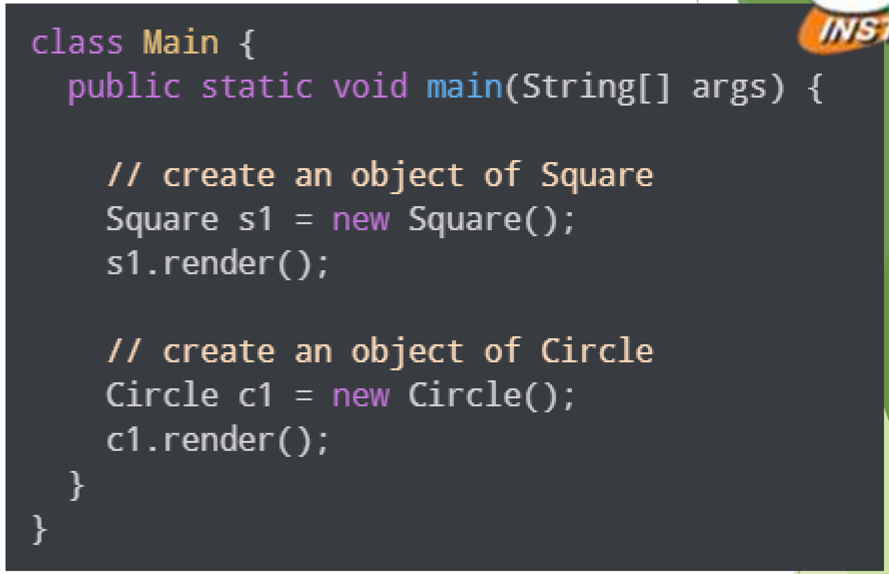
Polymorphism Example

- For example, given a base class ***shape***, polymorphism enables the programmer to define different area methods for any number of derived classes, such as ***circles, rectangles*** and ***triangles***.
- No matter what shape an object is, applying the area method to it will return the correct results.

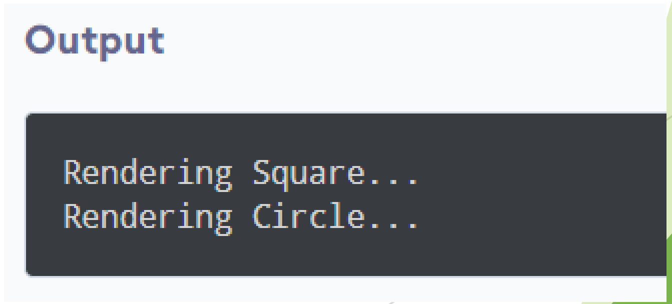
Java Polymorphism



```java
class Polygon {

  // method to render a shape
  public void render() {
    System.out.println("Rendering Polygon...");
  }
}

class Square extends Polygon {

  // renders Square
  public void render() {
    System.out.println("Rendering Square...");
  }
}

class Circle extends Polygon {

  // renders circle
  public void render() {
    System.out.println("Rendering Circle...");
  }
}
```



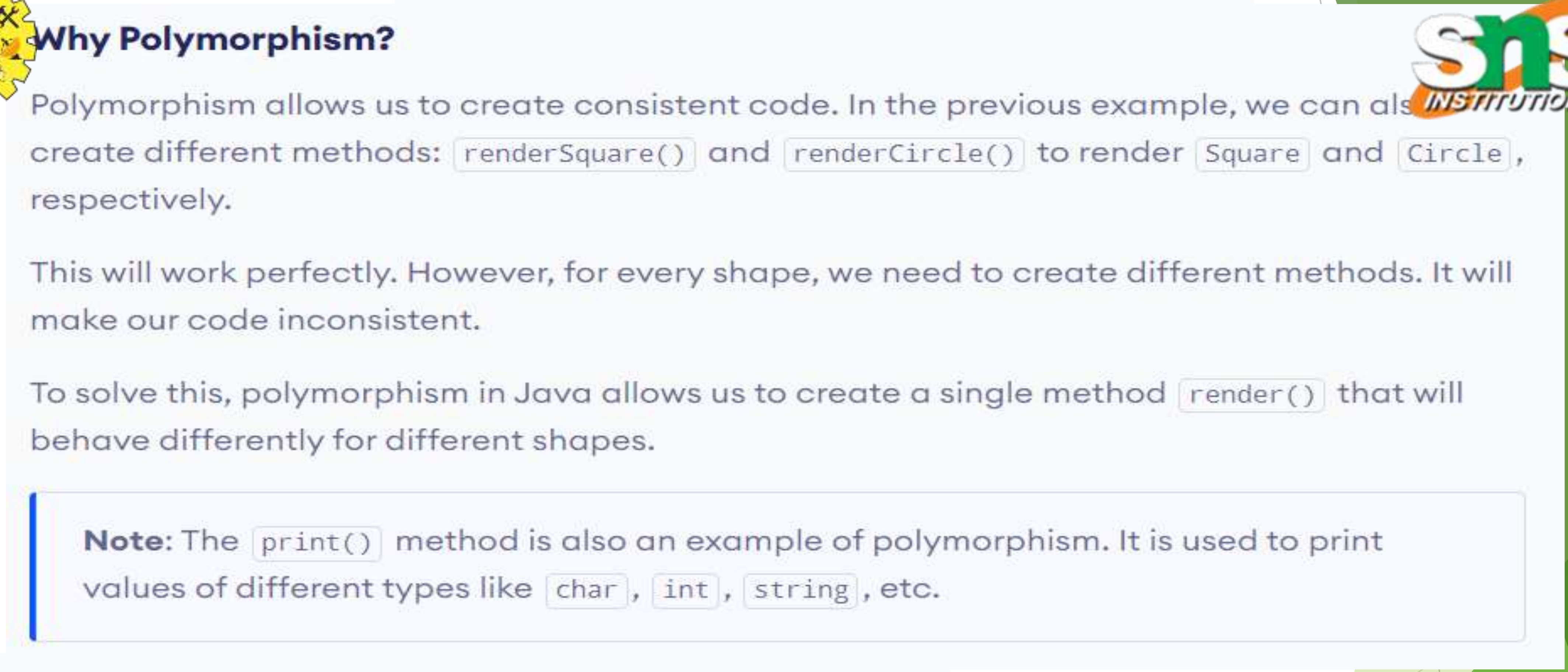
```java
class Main {
  public static void main(String[] args) {

    // create an object of Square
    Square s1 = new Square();
    s1.render();

    // create an object of Circle
    Circle c1 = new Circle();
    c1.render();
  }
}
```



Output

```
Rendering Square...
Rendering Circle...
```

Java Polymorphism



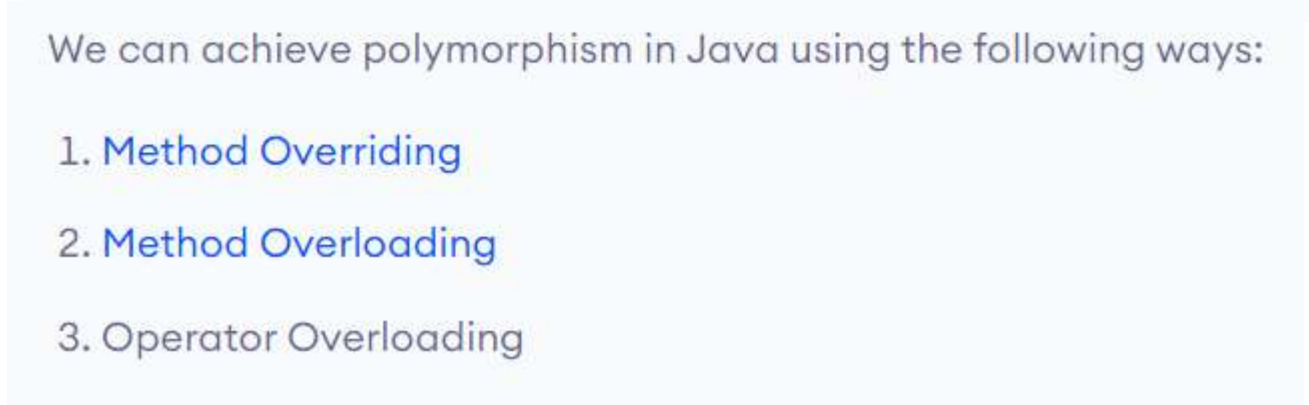
Why Polymorphism?

Polymorphism allows us to create consistent code. In the previous example, we can als create different methods: `renderSquare()` and `renderCircle()` to render `Square` and `Circle`, respectively.

This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.

To solve this, polymorphism in Java allows us to create a single method `render()` that will behave differently for different shapes.

> **Note:** The `print()` method is also an example of polymorphism. It is used to print values of different types like `char`, `int`, `string`, etc.



We can achieve polymorphism in Java using the following ways:

1. Method Overriding
2. Method Overloading
3. Operator Overloading

Java Polymorphism

Compile-Time Or Static Polymorphism

- Compile-time polymorphism refers to behaviour that is resolved when your Java class is compiled.
- Method overloading is an example of compile-time polymorphism. Method overloading is how you can use method with same name to do different things based on the parameters passed.

Illustration – Method Overloading

```
class MultiplyFun {
        int Multiply(int a, int b)
        {
                return a * b;
        }
        int Multiply(int a, int b, int c)
        {
                return a * b * c;
        }
}
class Main {
        public static void main(String[] args)
        {
                System.out.println(MultiplyFun.Multiply(2, 4));

                System.out.println(MultiplyFun.Multiply(2, 7, 3));
        }
}
```

Output:

```
8
42
```

Illustration – Method Overloading

```
class OperatorOVERDDN {

        void operator(String str1, String str2)
        {
                String s = str1 + str2;
                System.out.println("Concatinated String - "+ s);
        }
        void operator(int a, int b)
        {
                int c = a + b;
                System.out.println("Sum = " + c);
}       }
class Main {
        public static void main(String[] args)
        {
                OperatorOVERDDN obj = new
                OperatorOVERDDN();
                obj.operator(2, 3);
                obj.operator("joe", "now");
}       }
```

```
Output:
Sum = 5
Concatinated String - joenow
```

Java Polymorphism

Run-Time Or Dynamic Polymorphism

- Run-time polymorphism refers to behaviour that is resolved when your Java class is run by the JVM. Method overriding by the sub-class is an example of run-time polymorphism.
- Method overriding allows child classes to provide their own implementation of a method also defined in the parent class.
- The JVM decides which version of the method (the child's or the parent's) to call based on the object through which the method is invoked.

Illustration – Method Overriding

```
class Parent {
        void Print()
        {
          System.out.println("parent class");
        }
}
class subclass1 extends Parent {

        void Print()
        {
          System.out.println("subclass1");
        }
}
class subclass2 extends Parent {
        void Print()
        {
          System.out.println("subclass2");
        }
}
```

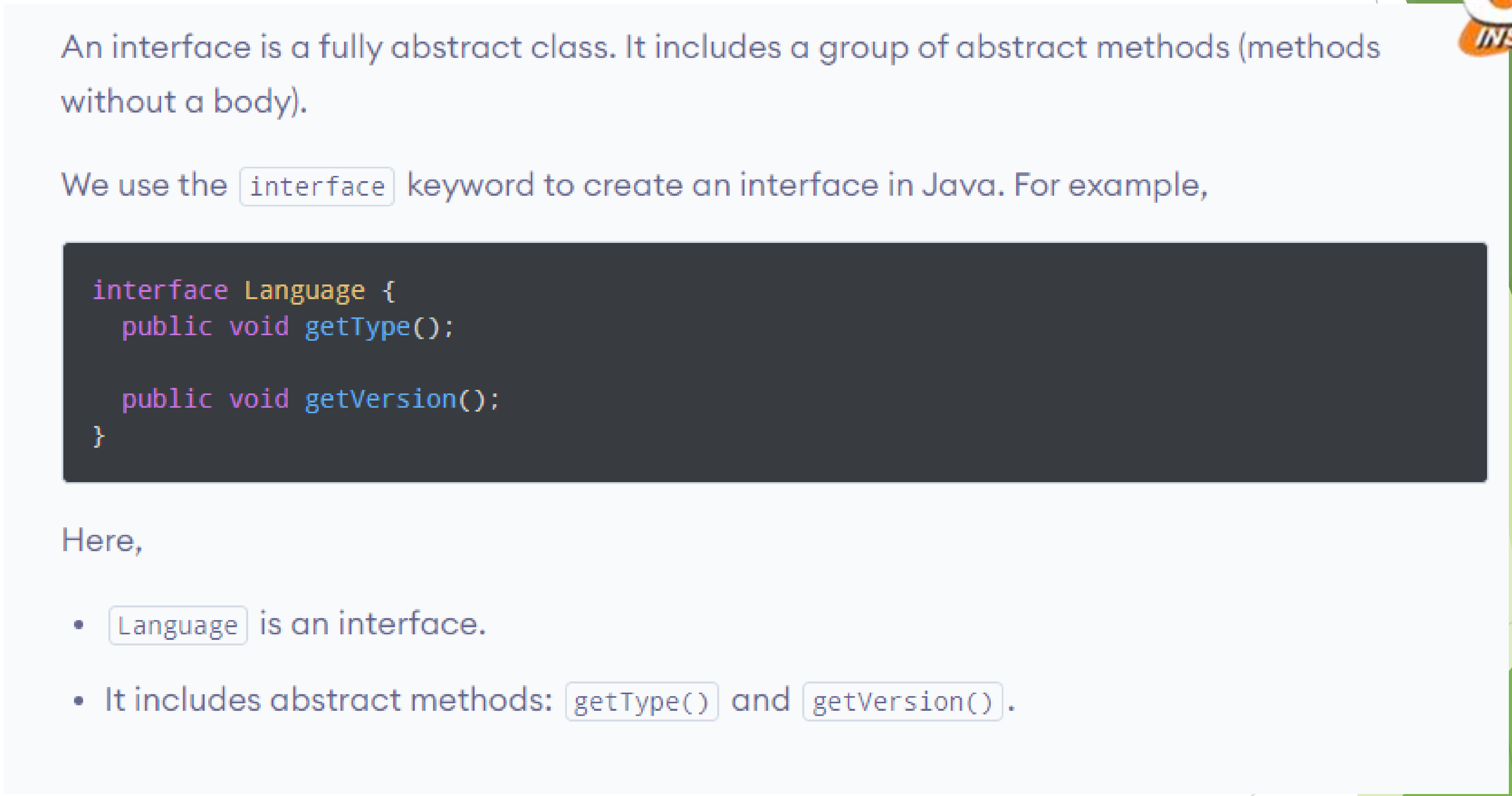
```
class TestPolymorphism3 {
        public static void main(String[] args)
        {
         Parent a;
         a = new subclass1();
         a.Print();
         a = new subclass2();
         a.Print();
        }
}
```

Output:
```
subclass1
subclass2
```

Advantages Of Polymorphism

- Code Cleanliness
- Ease Of Implementation
- Aligned With Real World
- Overloaded Constructors
- Reusability And Extensibility

Java Interface



An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

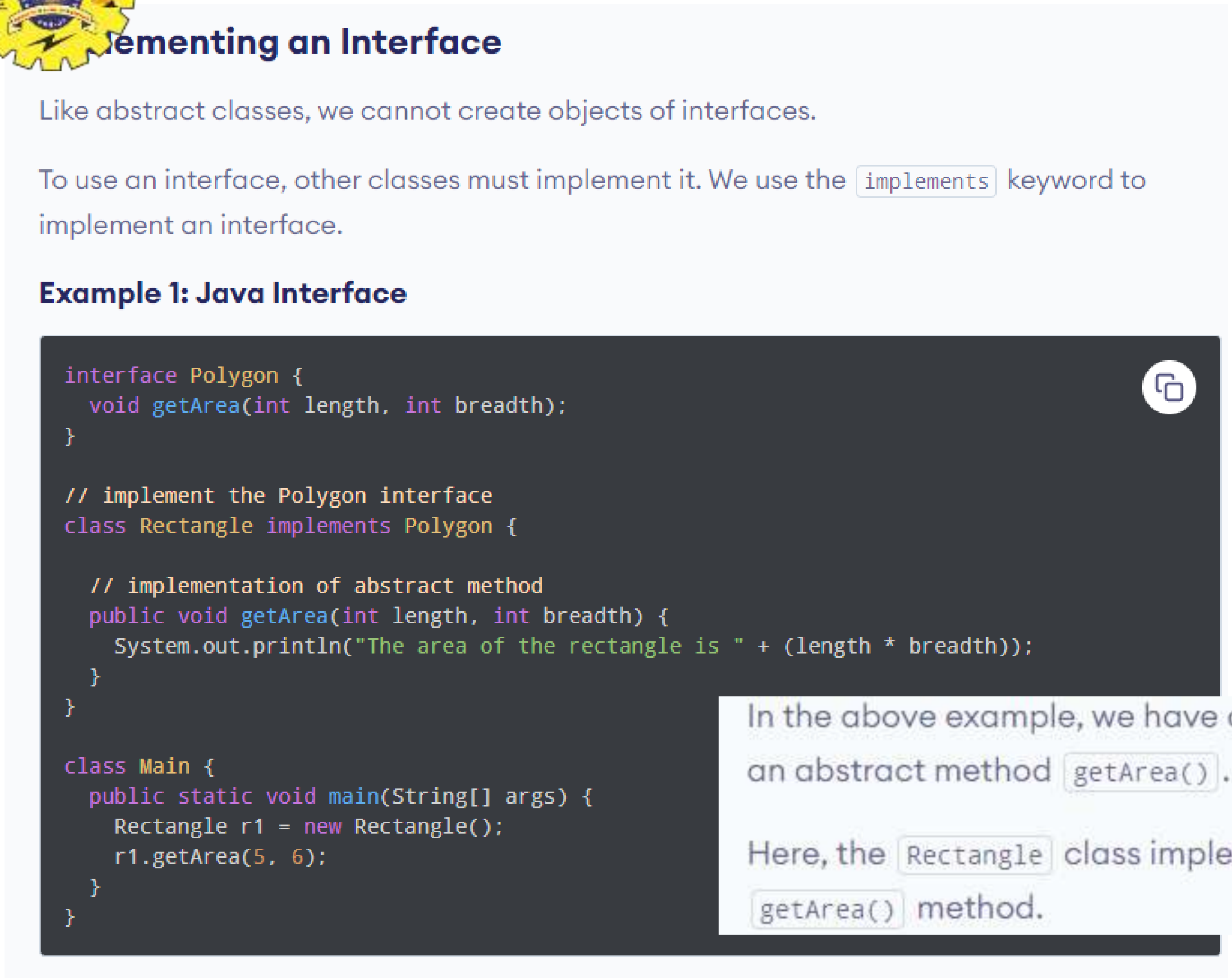
We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- `Language` is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

Java Interface



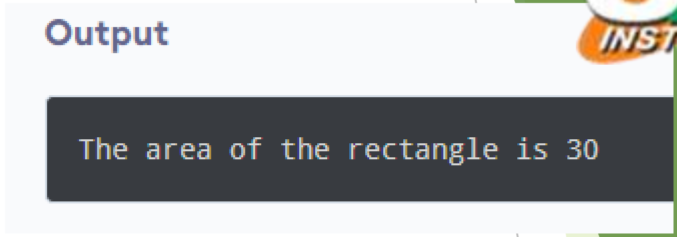
ementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

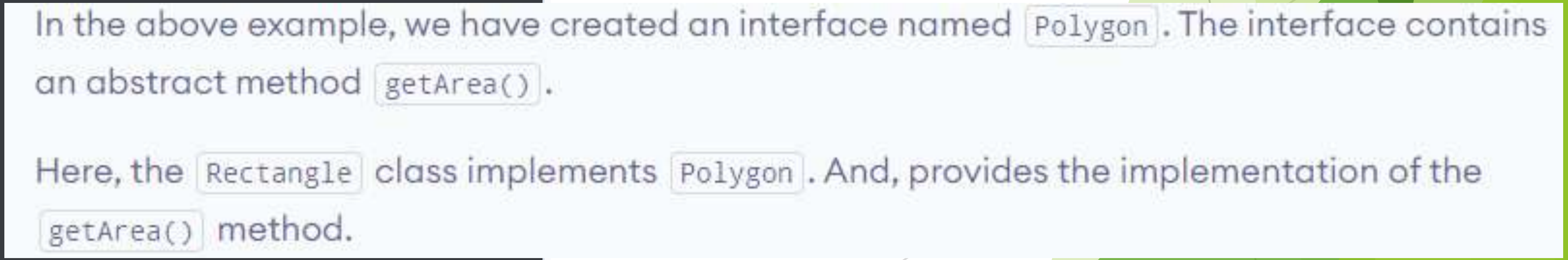
Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```



Output

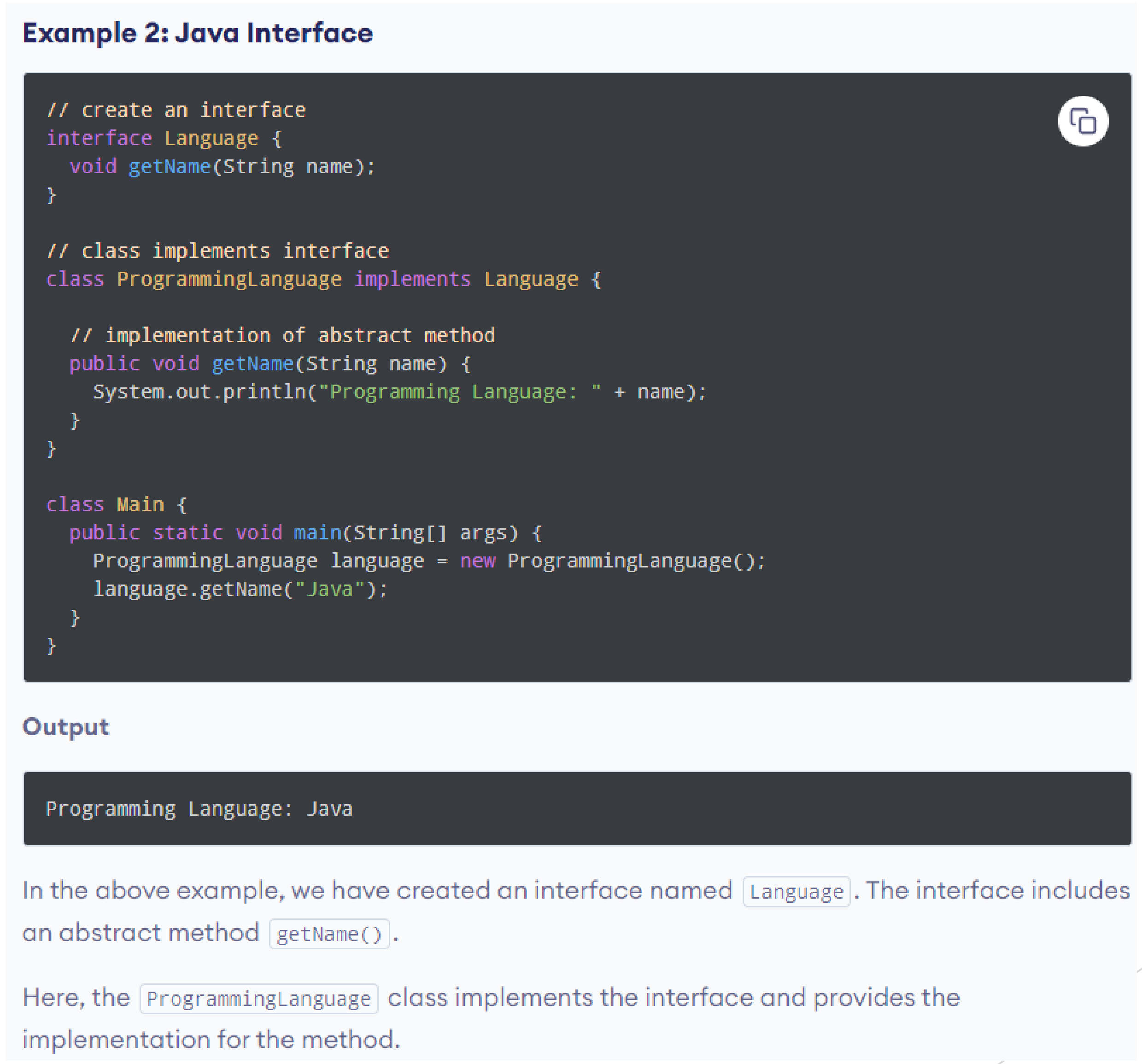
```
The area of the rectangle is 30
```



In the above example, we have created an interface named `Polygon`. The interface contains an abstract method `getArea()`.

Here, the `Rectangle` class implements `Polygon`. And, provides the implementation of the `getArea()` method.

Java Interface



Example 2: Java Interface

```java
// create an interface
interface Language {
  void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Output

```
Programming Language: Java
```

In the above example, we have created an interface named `Language`. The interface includes an abstract method `getName()`.

Here, the `ProgrammingLanguage` class implements the interface and provides the implementation for the method.

Java Interface

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Java Interface

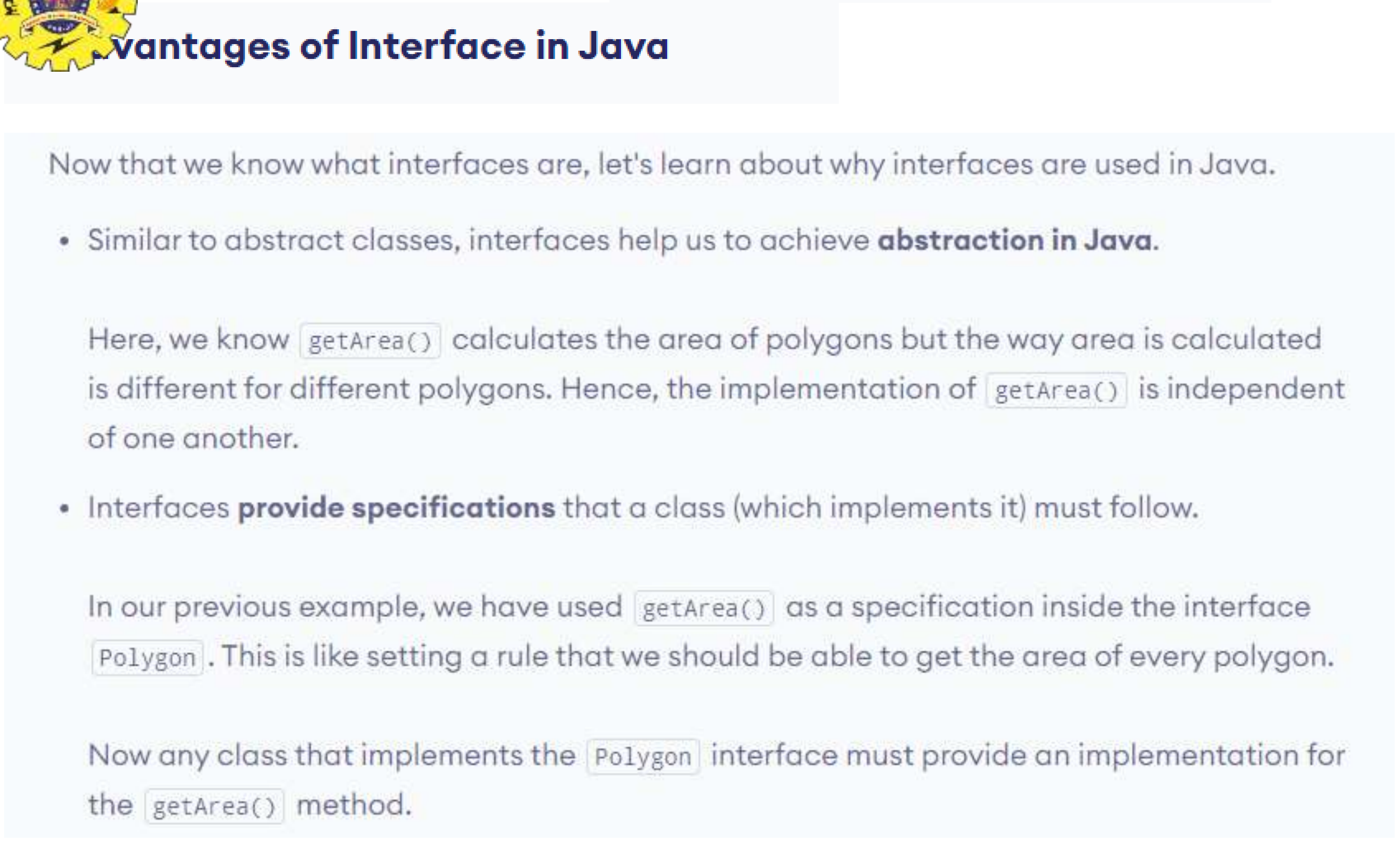
Extending an Interface

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the `Polygon` interface extends the `Line` interface. Now, if any class implements `Polygon`, it should provide implementations for all the abstract methods of both `Line` and `Polygon`.

Java Interface



vantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

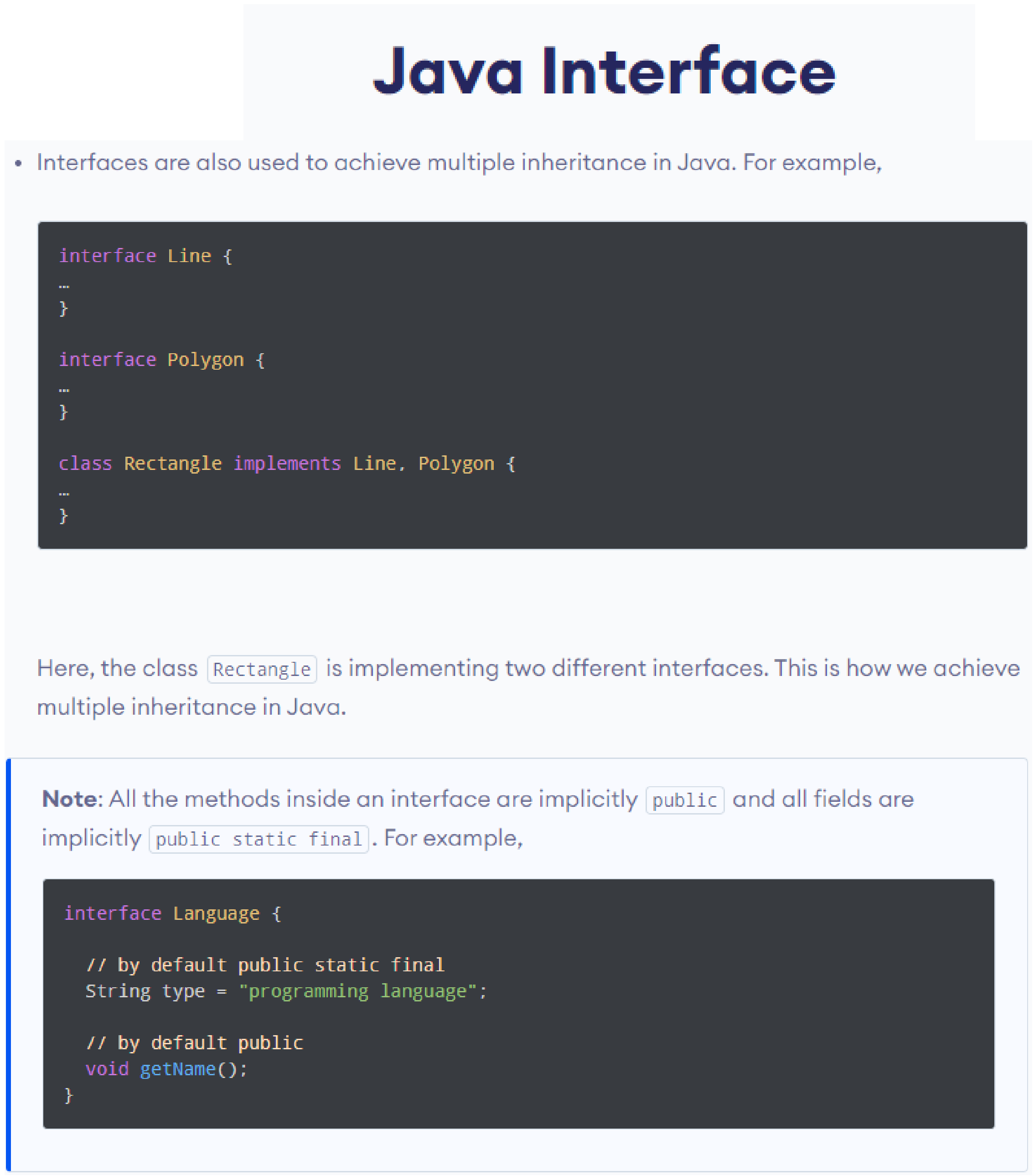
- Similar to abstract classes, interfaces help us to achieve **abstraction in Java.**

  Here, we know `getArea()` calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of `getArea()` is independent of one another.

- Interfaces **provide specifications** that a class (which implements it) must follow.

  In our previous example, we have used `getArea()` as a specification inside the interface `Polygon`. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the `Polygon` interface must provide an implementation for the `getArea()` method.



Java Interface

- Interfaces are also used to achieve multiple inheritance in Java. For example,

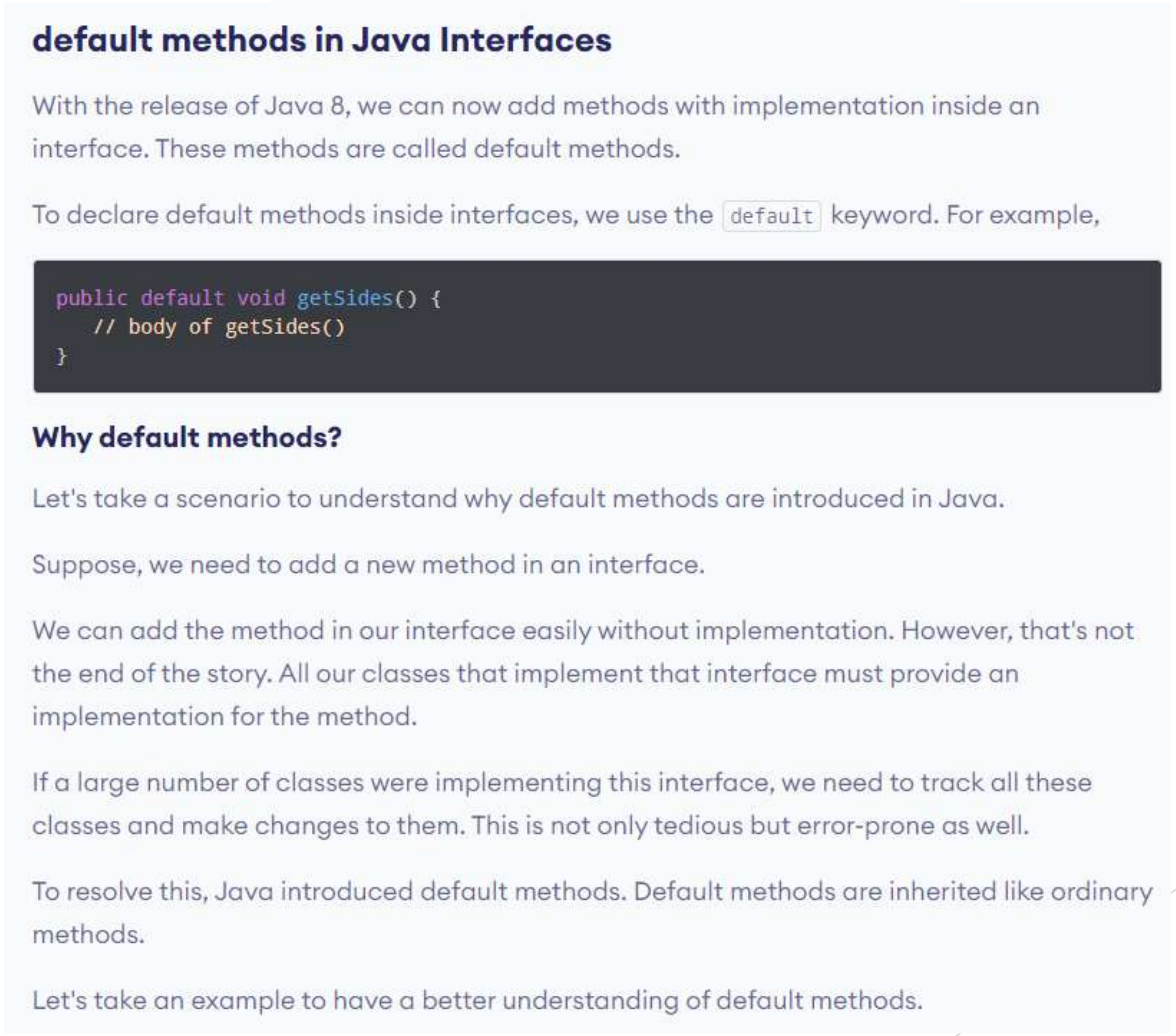
```
interface Line {
…
}

interface Polygon {
…
}

class Rectangle implements Line, Polygon {
…
}
```

Here, the class `Rectangle` is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

Note: All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,

```
interface Language {

  // by default public static final
  String type = "programming language";

  // by default public
  void getName();
}
```

Java Interface



default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the `default` keyword. For example,

```
public default void getSides() {
   // body of getSides()
}
```

Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface.

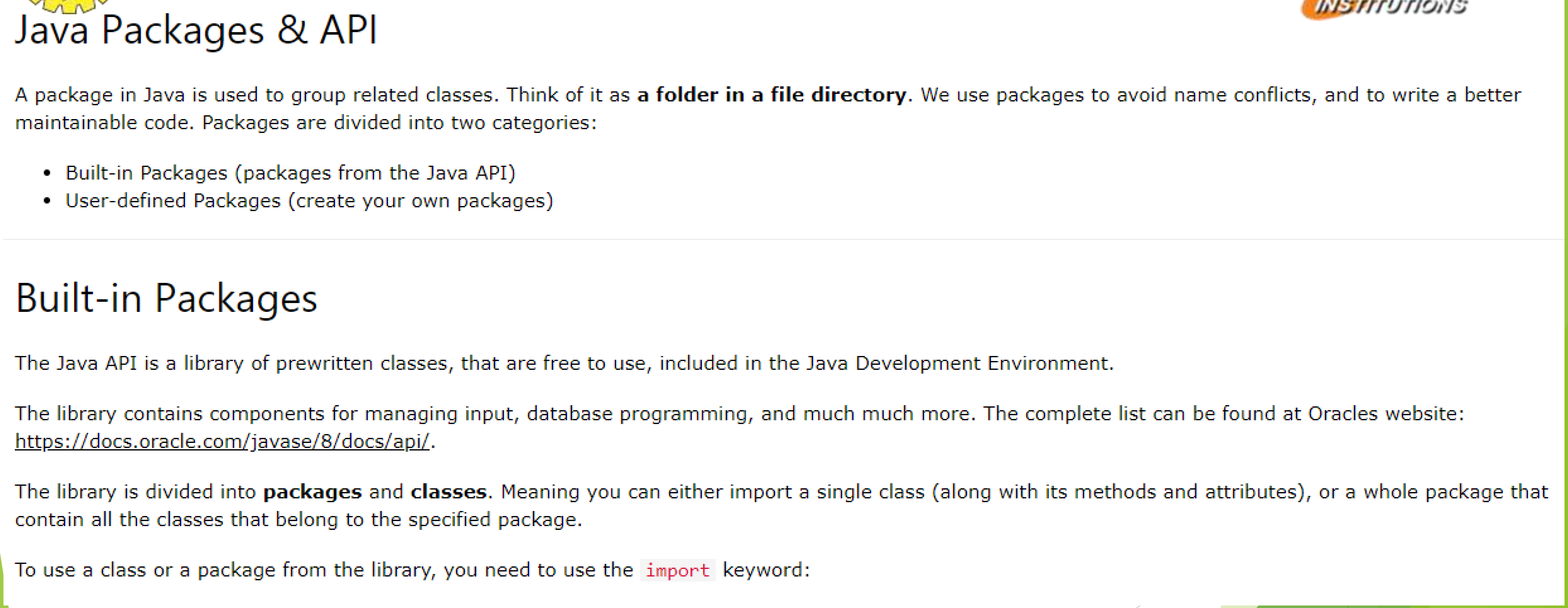
We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.

If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.

To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Let's take an example to have a better understanding of default methods.

Java Packages



Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

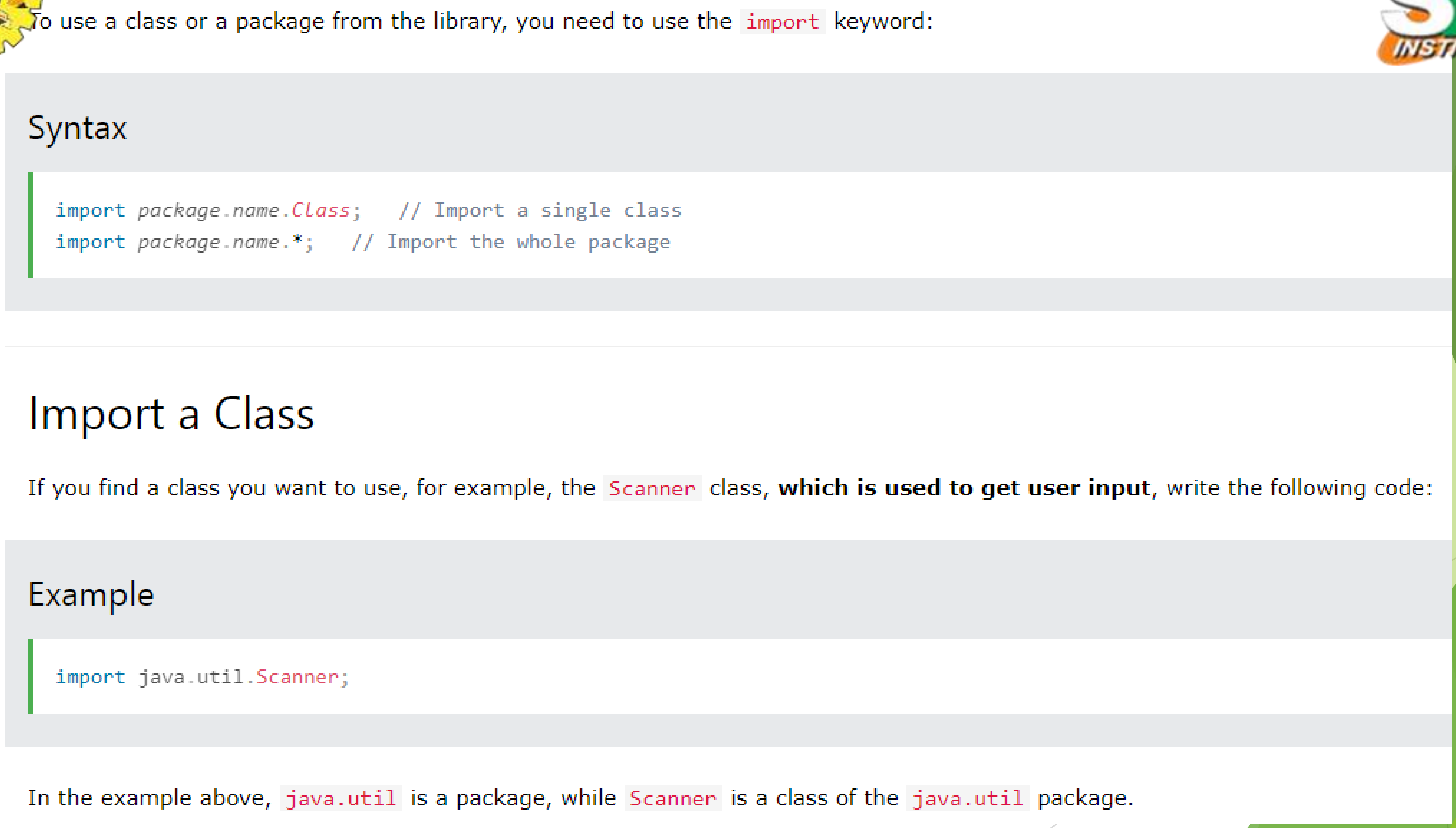
The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Java Packages



To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class

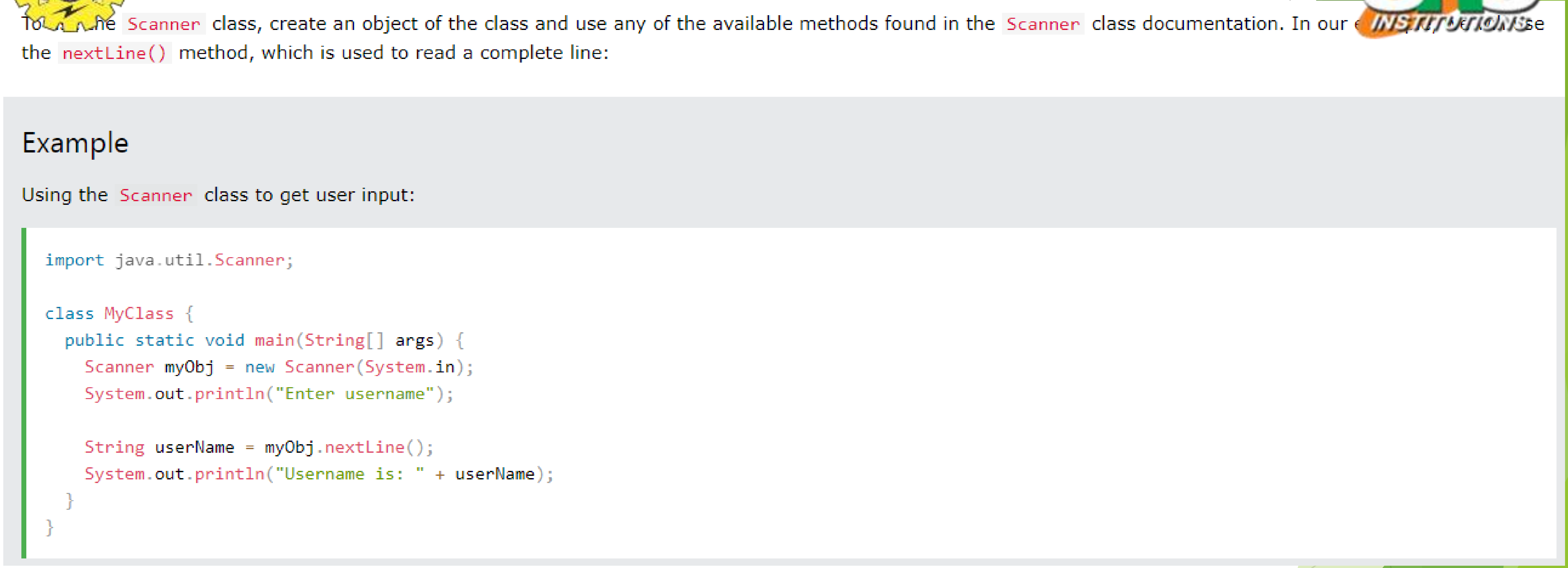
If you find a class you want to use, for example, the `Scanner` class, **which is used to get user input**, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

Java Packages



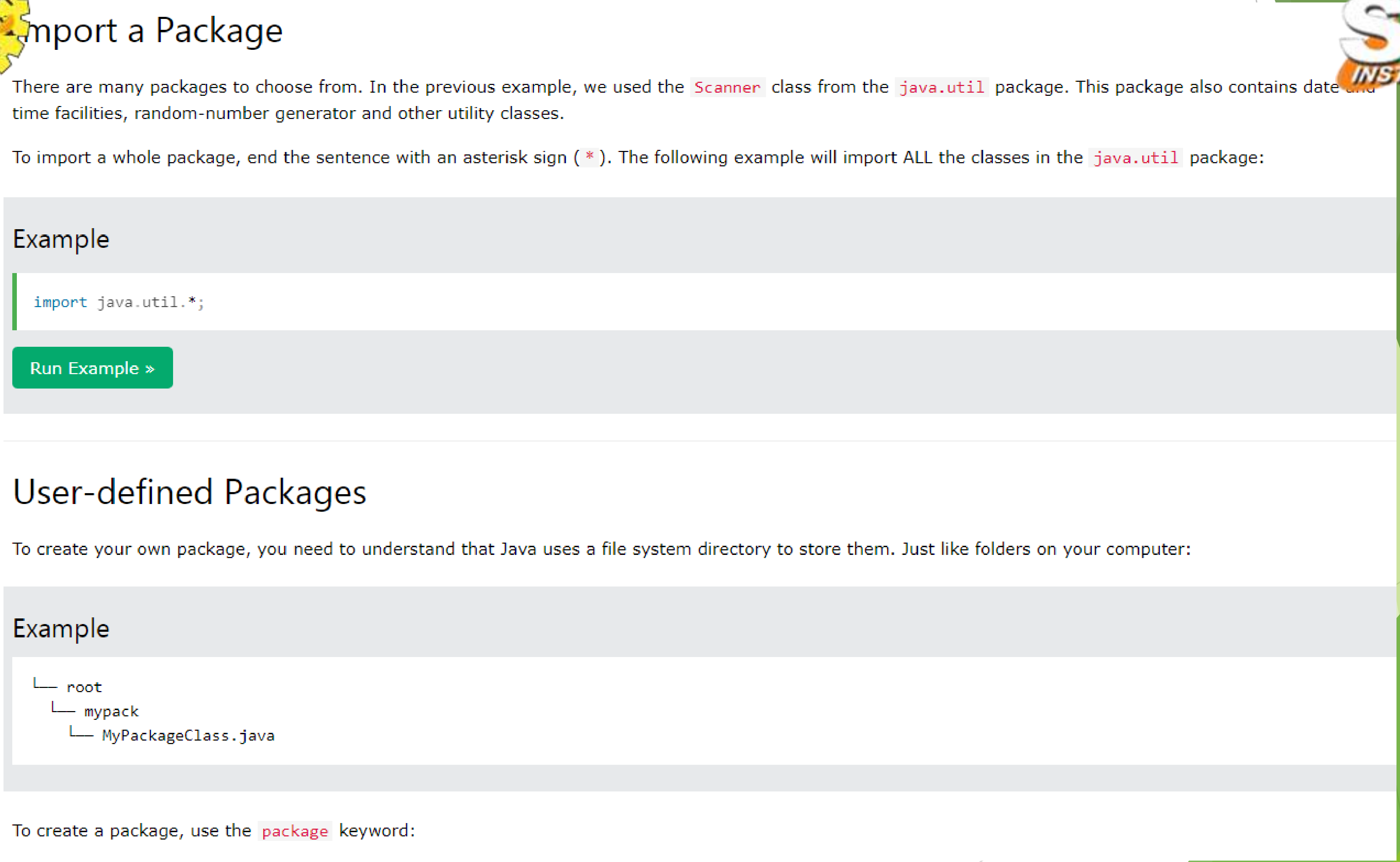
To [illegible] the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our e[illegible]se the nextLine() method, which is used to read a complete line:

Example

Using the Scanner class to get user input:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Java Packages



mport a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date
time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

Example

```
import java.util.*;
```

Run Example »

User-defined Packages

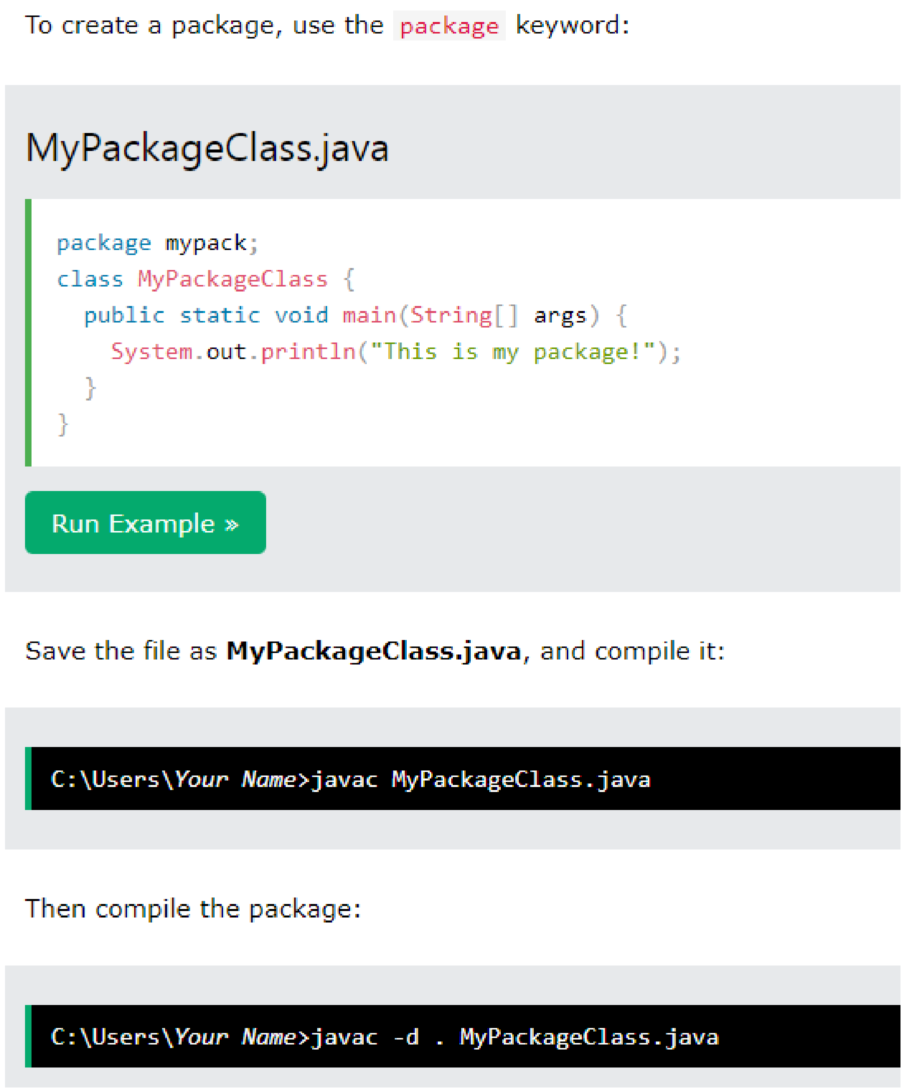
To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the `package` keyword:

Java Packages



To create a package, use the `package` keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

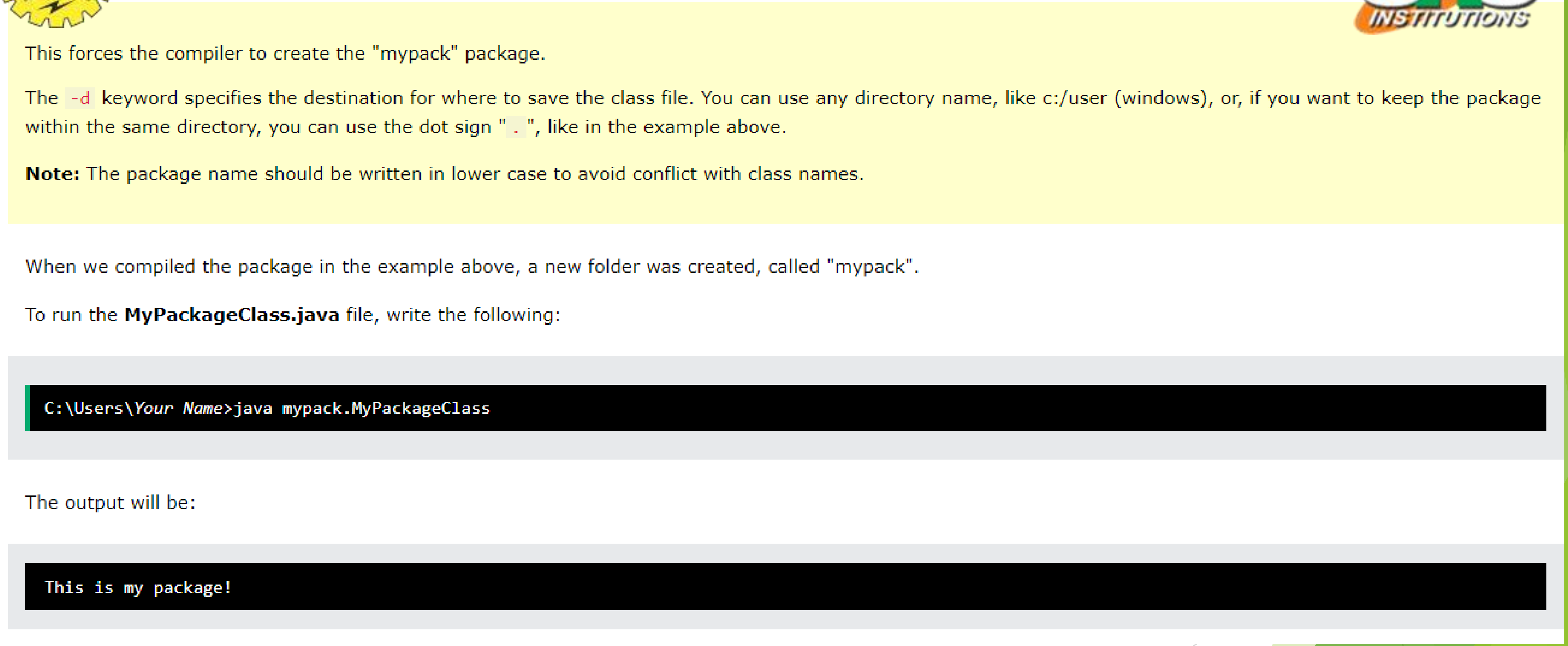
Save the file as **MyPackageClass.java**, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

Java Packages



This forces the compiler to create the "mypack" package.

The `-d` keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign "`.`", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```