Java Interface:

In Java, an interface is a way to define a contract or a set of abstract methods that a class must implement. It serves as a blueprint for classes that want to provide specific behavior without prescribing how that behavior should be implemented. Interfaces allow you to achieve abstraction and define a common set of methods that multiple classes can adhere to, promoting code reusability and flexibility in your codebase. Here are two code examples illustrating the concept of Java interfaces and their advantages:

Example 1: Basic Interface

```java
// Define an interface called "Drawable"
interface Drawable {
    void draw(); // Abstract method for drawing
}

// Implement the Drawable interface in a class
class Circle implements Drawable {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    // Implement the "draw" method as required by the interface
    @Override
    public void draw() {
        System.out.println("Drawing a circle with radius " + radius);
    }
}

class Rectangle implements Drawable {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Implement the "draw" method as required by the interface
    @Override
    public void draw() {
```

```
        System.out.println("Drawing a rectangle with width " + width + " and height " + height);
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Drawable circle = new Circle(5);
        Drawable rectangle = new Rectangle(4, 6);

        circle.draw(); // Calls the draw method of Circle class
        rectangle.draw(); // Calls the draw method of Rectangle class
    }
}
```

In this example, we have defined an interface `Drawable` with a single abstract method `draw()`. The `Circle` and `Rectangle` classes implement this interface by providing their own implementations of the `draw()` method. The `main` method demonstrates how you can use polymorphism to call the `draw` method on instances of both classes, even though they have different implementations.

Example 2: Interface for Multiple Inheritance

```java
// Define two interfaces
interface Walkable {
    void walk();
}

interface Swimmable {
    void swim();
}

// Implement the interfaces in a class
class Human implements Walkable {
    @Override
    public void walk() {
        System.out.println("A human is walking.");
    }
}

class Fish implements Swimmable {
    @Override
    public void swim() {
```

```
        System.out.println("A fish is swimming.");
    }
}

// Create a class that implements multiple interfaces
class Athlete implements Walkable, Swimmable {
    @Override
    public void walk() {
        System.out.println("An athlete is walking.");
    }

    @Override
    public void swim() {
        System.out.println("An athlete is swimming.");
    }
}

public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Human person = new Human();
        Fish fish = new Fish();
        Athlete athlete = new Athlete();

        person.walk();
        fish.swim();
        athlete.walk();
        athlete.swim();
    }
}
```

In this example, we have two interfaces `Walkable` and `Swimmable`, each with a single abstract method. The `Human` class implements `Walkable`, the `Fish` class implements `Swimmable`, and the `Athlete` class implements both `Walkable` and `Swimmable`. This demonstrates how Java interfaces allow a class to implement multiple interfaces, enabling a form of multiple inheritance in Java without the complications associated with multiple inheritance through classes.

Advantages of Java Interfaces:

1. **Abstraction**: Interfaces allow you to define a contract for classes to follow, promoting abstraction and separation of concerns in your code.

2. **Multiple Inheritance**: Java interfaces enable a class to implement multiple interfaces, allowing for the inheritance of behavior from multiple sources without the complexities of multiple inheritance through classes.

3. **Polymorphism**: Interfaces facilitate polymorphism, where different classes can be treated as instances of the same interface, providing flexibility and code reusability.

4. **API Design**: Interfaces are often used in designing APIs, ensuring that classes implementing the interface adhere to a specific set of rules and behaviors.

5. **Code Organization**: Interfaces help in organizing your code by defining a clear contract, making it easier to understand and maintain.

6. **Testing and Mocking**: Interfaces make it easier to create mock objects for testing, as you can create mock implementations of interfaces to isolate and test specific parts of your code.

7. **Dynamic Behavior**: Interfaces allow you to achieve dynamic behavior by implementing them at runtime, making your code more adaptable to different scenarios.