Lambda expressions in Java are a powerful feature introduced in Java 8 that allows you to define and use anonymous functions. Lambda expressions provide a concise way to represent a block of code that can be passed around as data. They are particularly useful when working with functional interfaces, which have a single abstract method (SAM), such as Java's built-in `Runnable`, `Comparator`, or user-defined functional interfaces.

Here are some advantages of using Java lambda expressions:

1. **Concise Syntax:** Lambda expressions allow you to write shorter and more readable code, reducing the verbosity of anonymous inner classes.

2. **Readability:** They make your code more readable by expressing the intention of the code more clearly.

3. **Flexibility:** Lambdas can be used in various contexts, such as passing behavior to methods or defining event handlers.

4. **Functional Programming:** They facilitate functional programming paradigms by treating functions as first-class citizens.

Here are some code examples demonstrating the advantages of Java lambda expressions:

Example 1: Using Lambda Expression with `Runnable`

```java
// Without Lambda
Runnable runnable1 = new Runnable() {
    public void run() {
        System.out.println("Hello from Runnable!");
    }
};

// With Lambda
Runnable runnable2 = () -> {
    System.out.println("Hello from Lambda Runnable!");
};

// Execute the run method
runnable1.run();
runnable2.run();
```

In this example, we create two `Runnable` instances, one using an anonymous inner class and the other using a lambda expression. The lambda expression version is more concise and easier to read.

Example 2: Using Lambda Expression with `Comparator`

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Sorting with Anonymous Inner Class
Collections.sort(names, new Comparator<String>() {
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
});

// Sorting with Lambda Expression
Collections.sort(names, (a, b) -> a.compareTo(b));

System.out.println(names);
```

In this example, we use a lambda expression to simplify the sorting of a list of strings. The lambda expression version eliminates the need to create a separate `Comparator` implementation.

Example 3: Using Lambda Expression with Functional Interfaces

```java
// Custom functional interface
interface MathOperation {
    int operate(int a, int b);
}

public static void main(String[] args) {
    MathOperation add = (a, b) -> a + b;
    MathOperation subtract = (a, b) -> a - b;
    MathOperation multiply = (a, b) -> a * b;

    System.out.println("Addition: " + operate(10, 5, add));
    System.out.println("Subtraction: " + operate(10, 5, subtract));
    System.out.println("Multiplication: " + operate(10, 5, multiply));
}
```

```java
static int operate(int a, int b, MathOperation operation) {
    return operation.operate(a, b);
}
```

In this example, we define a custom functional interface `MathOperation` and use lambda expressions to create instances of it. Then, we pass these lambda expressions as arguments to the `operate` method, demonstrating how lambdas can be used to pass behavior as data.

Lambda expressions in Java are a powerful tool for simplifying code and promoting a more functional style of programming, especially when working with functional interfaces. They make the code more readable and expressive.