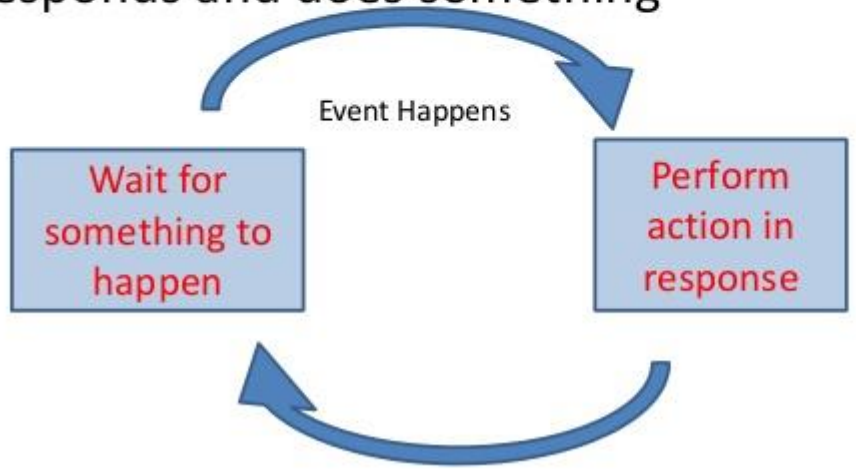
Event Driven Programming in Java

Event-driven programming

- The flow of the program is determined by events
- It is the dominant paradigm used in graphical user interfaces and web applications
- Centered on performing certain actions in response to user input
- Events such as:
  - user actions
  - mouse clicks
  - key presses
  - sensor outputs
  - messages from other programs/threads

Event driven programming

- Program waits for events
- Whenever something happens the program responds and does something



A simple Event Listener Interface

```
public interface EventListener {

    public void onSomeChange(State oldState, State newState);

}
```

A simple class using the listener Interface

```
public class EventOwner {

    public void addEventListener(EventListener listener) { ... }

}
```

Implementation in Java 7

Anonymous Interface Implementation!

```
EventOwner eventOwner = new EventOwner();

eventOwner.addEventListener(new EventListener() {

    public void onSomeChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```

Implementation in Java 8

Lambda expression usage

- The lambda expression is matched against the parameter type of the addEventListener() method's parameter
- If the lambda expression matches the parameter type (in this case the EventListener interface) , then the lambda expression is turned into a function that implements the same interface as that parameter.

Matching Lambdas and Interfaces

- A single method interface is also sometimes referred to as a functional interface
- We have to follow 3 rules
  - The interface should have only one method
  - The parameters of the lambda expression should match the parameters of the single method
  - The return type of the lambda expression should match the return type of the single method

Lambda Expressions with Zero Parameters

```
() -> System.out.println("Zero parameter lambda");
```

Lambda Expressions with One Parameter

```
(param) -> System.out.println("One parameter: " + param);

or

param -> System.out.println("One parameter: " + param);
```

Lambda Expressions with Multiple Parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Lambda Expression Parameter Types

- Specifying parameter types for a lambda expression may be necessary if the compiler cannot infer the parameter types from the functional interface method the lambda expression is matching

```
(Student student1) -> System.out.println("Student's name is: " + student1.getName());
```

Lambda Expression Function Body One Line

```
(oldState, newState) -> System.out.println("Something changed!")
```

Lambda Expression Function Body Multiple Lines

```
(oldState, newState) -> {
    System.out.println("Old state: " + oldState);
    System.out.println("New state: " + newState);
}
```

Lambda Expression Returning Value

```
(param) -> {
    System.out.println("param: " + param);
    return "some value";
}
```

Lambda Expressions as Objects

```
public interface MyComparator {

    public boolean compare(int a1, int a2);

}

MyComparator myComparator = (a1,a2) -> {return a1 > a2;};

boolean result = myComparator.compare(5, 10);
```