2023-03-10

# Interfaces

# Interfaces

Languages like Python and C++ support multiple inheritance: Classes can inherit from multiple base classes.

Java has single inheritance. A class can only declare a single superclass.

However, Java has another feature called **interfaces**, which is similar to multiple inheritance.

A class `extends` only one superclass, but it `implements` zero or more **interfaces**.

> Interfaces used to be on the AP Computer Science exam, but were removed in 2017. So, consider this bonus content.

# Abstract classes are "incomplete"

MazeObject is an **abstract class** – it cannot be instantiated itself. It has **abstract methods** which must be implemented by a subclass to "complete" the class, like getImagePath.

```
// Subclasses must override getImagePath to return the path of the image file to use.
public abstract String getImagePath();

// Subclasses must override getName to return a descriptive name.
public abstract String getName();

// Subclasses may override this to indicate whether light cannot pass through.
// (Wall, for instance, overrides this to return true.)
public boolean isOpaque() { return false; }

// Subclasses may override this to provide per-tick behavior, such as movement.
public void tick() {}

// Subclasses may override this to provide interactive behavior. The status to be displayed
// should be returned, or null if none.
public String interact() { return null; }
```

# Interfaces

An interface declaration looks much like a class declaration.

The big difference is the methods have no bodies.

This is similar to **abstract methods** like MazeObject.getImagePath.

```
/**
 * The listener interface for receiving KeyEvents.
 */
public interface KeyListener extends EventListener {
    /**
     * KEY_PRESSED events are fired when any key (including a function
     * key and cursor key) is pressed while the component has keyboard
     * input focus.
     * KeyEvent.getKeyCode() can be used to find out which key was pressed.
     */
    void keyPressed(KeyEvent ke);

    /**
     * KEY_TYPED events are fired when a key representing a valid text
     * character (not a function key or cursor key) is pressed.
     * KeyEvent.getKeyChar() can be used to get the ASCII code of the key
     * that was pressed.
     */
    void keyTyped(KeyEvent ke);

    /**
     * KEY_RELEASED events are fired when a key is released.
     */
    void keyReleased(KeyEvent ke);
}
```

# Interfaces

Interfaces are like abstract classes, except ones where **every** method is abstract.*

An interface is a contract that a class has to implement completely.

If a class implements KeyListener, it must implement keyPressed, keyTyped and keyReleased, or it's a compile error.

**Mostly true. Default interface methods added in Java 8 (2014)*

```
/**
 * The listener interface for receiving KeyEvents.
 */
public interface KeyListener extends EventListener {
    /**
     * KEY_PRESSED events are fired when any key (including a function
     * key and cursor key) is pressed while the component has keyboard
     * input focus.
     * KeyEvent.getKeyCode() can be used to find out which key was pressed.
     */
    void keyPressed(KeyEvent ke);

    /**
     * KEY_TYPED events are fired when a key representing a valid text
     * character (not a function key or cursor key) is pressed.
     * KeyEvent.getKeyChar() can be used to get the ASCII code of the key
     * that was pressed.
     */
    void keyTyped(KeyEvent ke);

    /**
     * KEY_RELEASED events are fired when a key is released.
     */
    void keyReleased(KeyEvent ke);
}
```

# Game implements interfaces

Look at Game.java. Note that it doesn't declare any superclass, so the superclass is Object. It implements two interfaces, though: KeyListener and ActionListener.

```
public class Game implements KeyListener, ActionListener {
  private JFrame frame;
  private JLabel statusLine;
  private Maze maze;
  private MazeView mazeView;
  private Player player;
  private Timer timer;
```

# Game's KeyListener methods

Because Game implements KeyListener, it must implement keyPressed, keyTyped, and keyReleased methods. We only cared about keyPressed, but we had to supply something for the other two.

```
@Override
public void keyPressed(KeyEvent event) {
  int keyCode = event.getKeyCode();
  if (keyState == NORMAL_KEY_STATE) {
    if (keyCode == KeyEvent.VK_LEFT) {
      movePlayerBy(-1, 0);
    } else if (keyCode == KeyEvent.VK_RIGHT) {
      movePlayerBy(1, 0);
    } else if (keyCode == KeyEvent.VK_UP) {
      movePlayerBy(0, -1);
    } else if (keyCode == KeyEvent.VK_DOWN) {
      movePlayerBy(0, 1);
    } else if (keyCode == KeyEvent.VK_Q) {
      keyState = CONFIRM_QUIT_STATE;
      statusLine.setText("Are you sure you want to quit? (Y/N)");
    }
```

```
@Override
public void keyTyped(KeyEvent event) {
}

@Override
public void keyReleased(KeyEvent event) {
}
```

- There is a fancy-pants way to avoid the empty methods, an abstract class called KeyAdapter + anonymous inner classes.
- Java 8 (2014) did add "default interface methods" (which have bodies)

# Game's call to addKeyListener

Game registers itself with Java Swing as a key listener by calling addKeyListener on the JFrame that is the game's main window.

The addKeyListener method takes a parameter of type KeyListener. Interfaces are types!

Because Game implements KeyListener, it can be cast to KeyListener.

```
// Legal.
KeyListener k = (KeyListener)game;
```

```
public Game() {
  maze = new Maze(this);

  statusLine = new JLabel();
  statusLine.setFont(new Font("Serif", Font.PLAIN, 24));
  statusLine.setText("Welcome to ElCoRogue!");

  mazeView = new MazeView(maze);

  frame = new JFrame("Maze");
  frame.getContentPane().setLayout(new BorderLayout());
  frame.getContentPane().add(mazeView, BorderLayout.CENTER);
  frame.getContentPane().add(statusLine, BorderLayout.NORTH);
  frame.addKeyListener(this);
  frame.setSize(800, 800);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setLocationRelativeTo(null);
  frame.setExtendedState(JFrame.MAXIMIZED_BOTH);
  frame.setVisible(true);

  timer = new Timer(250, this);
  timer.start();
}
```

# Game's ActionListener

The game "ticks" every 1/4 second, giving every MazeObject a chance to do something.

This is done using a class called java.swing.Timer.

You create a Timer, and register an ActionListener with it. The ActionListener's actionPerformed method will be called every time the timer goes off. (The 250 is milliseconds for 1/4 of a second.)

```
public Game() {
  maze = new Maze(this);

  statusLine = new JLabel();
  statusLine.setFont(new Font("Serif", Font.PLAIN, 24));
  statusLine.setText("Welcome to ElCoRogue!");

  mazeView = new MazeView(maze);

  frame = new JFrame("Maze");
  frame.getContentPane().setLayout(new BorderLayout());
  frame.getContentPane().add(mazeView, BorderLayout.CENTER);
  frame.getContentPane().add(statusLine, BorderLayout.NORTH);
  frame.addKeyListener(this);
  frame.setSize(800, 800);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setLocationRelativeTo(null);
  frame.setExtendedState(JFrame.MAXIMIZED_BOTH);
  frame.setVisible(true);

  timer = new Timer(250, this);
  timer.start();
}
```

# Game's ActionListener.actionPerformed method

**Timer**

```
public Timer(int delay,
             ActionListener listener)
```

Creates a `Timer` and initializes both the initial delay and between-event delay to `delay` milliseconds. If `delay` is less than or equal to zero, the timer fires as soon as it is started. If `listener` is not `null`, it's registered as an action listener on the timer.

**Parameters:**

`delay` - milliseconds for the initial and between-event delay

`listener` - an initial listener; can be `null`

**See Also:**

`addActionListener(java.awt.event.ActionListener)`,
`setInitialDelay(int)`,
`setRepeats(boolean)`

```
@Override
public void actionPerformed(ActionEvent event) {
  if (playing) {
    maze.tick();
    mazeView.repaint();
  }
}
```

```
public interface ActionListener extends EventListener
{
  /**
   * This method is invoked when an action occurs.
   *
   * @param event the <code>ActionEvent</code> that occurred
   */
  void actionPerformed(ActionEvent event);
}
```

# List and ArrayList

You know how you occasionally see code like this?

```
List<String> list = new ArrayList<String>();
```

So, List must be a superclass of ArrayList, right?

Actually, List is an interface which ArrayList implements!

List even has *superinterfaces* which are like superclasses.

java.util

**Interface List<E>**

**Type Parameters:**

E - the type of elements in this list

**All Superinterfaces:**

Collection<E>, Iterable<E>

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

# Iterable<E>

ArrayList<E> implements the interface List<E>

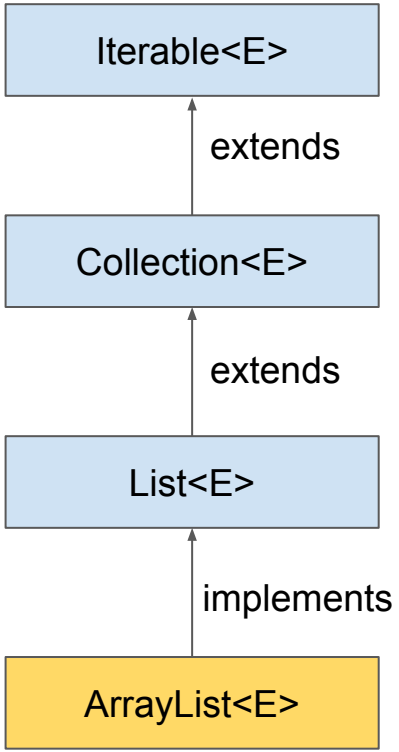
List<E> extends Collection<E>

Collection<E> extends Iterable<E>

That is, Iterable<E> is a superinterface of List<E>

The for-each loop works on any object that implements Iterable<E>

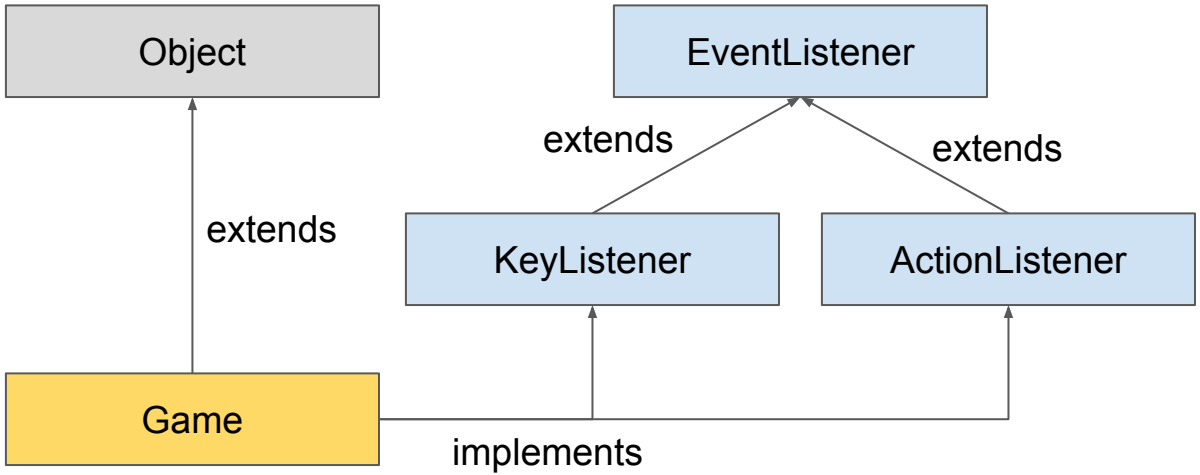
This is why for-each loops work on ArrayLists!



# Class Hierarchy / Interface Hierarchy

A class belongs to an inheritance hierarchy of classes. When a class implements interfaces, the class is also part of an inheritance hierarchy of interfaces.

```
public class Game implements KeyListener, ActionListener {
```




```
Game game = new Game();
```

These are all true:

```
game instanceof Object
game instanceof EventListener
game instanceof KeyListener
game instanceof ActionListener
```

# Interfaces

In conclusion:

You don't need to study them for the AP exam, but interfaces are widely used in Java code.

Frameworks that you build applications with, like Java Swing, use them a lot.

Even the Java standard library (ArrayList and friends) uses interfaces.

So, good to know!