Event Handling
Java 7

Waterford Institute of Technology

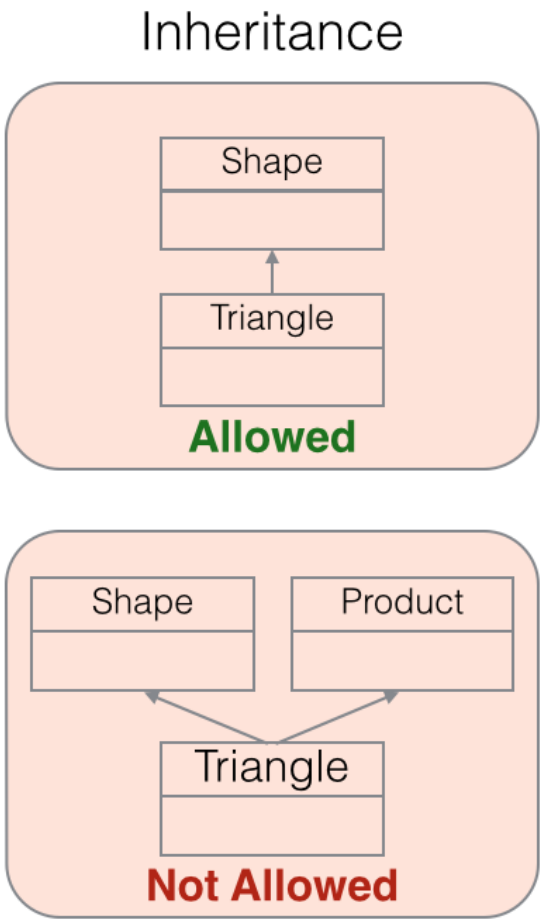
September 23, 2014

John Fitzgerald

Inheritance

Inheritance v Interface

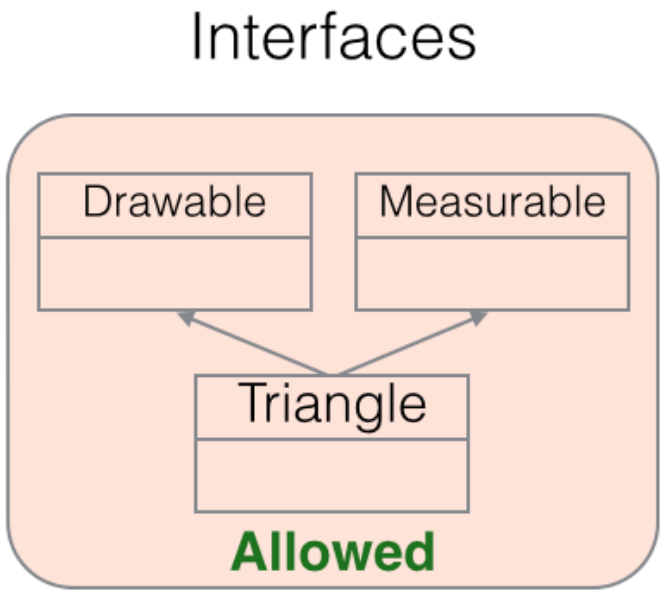
- Inheritance rather than interfaces?
  - Complexity: simpler to use interfaces
  - Class can inherit only from one class
  - Class can implement many interfaces



Inheritance

Inheritance v Interface

- Class may implement several interfaces?
  - Class Triangle must implement all methods in interfaces Drawable and Measurable
  - Rules to determine which method to implement in case of name clashes



Java *interface*

A subset

interface is a Java type that may contain only

- Method signatures
- Constant declarations

Note that

- *interface* defines interfaces
- *class* defines classes
- Methods implemented in class that implements interface

```
public interface Drawable
{
    public void draw();
    public void scale(int x, int y);
}
```

access modifier public optional

Java *interface*

Compare with *class*

Java *interface* different from *class*

- *interface* specifies behaviour only
- Cannot create objects of an *interface*
- Create objects of classes that implement interfaces

```
public class Tree implements Drawable
{
    public void draw() {
        ...
    }
}

Tree tree = new Tree();
tree.draw();
```

Java *interface*

Implementation

A class may:

- Provide additional methods unrelated to interface
- Is obliged to implement all methods in interface
- May, optionally, provide @Override annotation to implemented methods

```
public class Triangle implements Drawable
{
   @Override
   public void draw() {...}//must implement draw
   @Override
   public void scale(int x, int y) {...}//must implement scale
   public int getArea(){...}//may include additional methods
}
```

Java *interface*

Implementation

Many classes may implement particular interface

- Class states that it implements particular interface

  public class `Triangle` implements `Drawable` {. . . }

- Class provides suitable implementation of interface methods

```
public class Triangle implements Drawable
{
   @Override
   public void draw() {...}
}

public class House implements Drawable
{
   @Override
   public void draw() {...}
}
```

Java *interface*

Application

Object of class implementing interface may be stored in variable whose type is the interface

- Tree implements Drawable
- Tree object reference can be stored in Drawable variable
- Facilitates unifying behaviour

```
Drawable element = new Tree(...); /*legal*/
```

```
//create array of Drawable variables
Drawable[] elements = new Drawable[2];
//Assign different objects to elements in array
Drawable elements[0] = new House(...);
Drawable elements[1] = new Triangle(...);
```

Java *interface*

What you cannot do

- Illegal to attempt instantiation of interface.

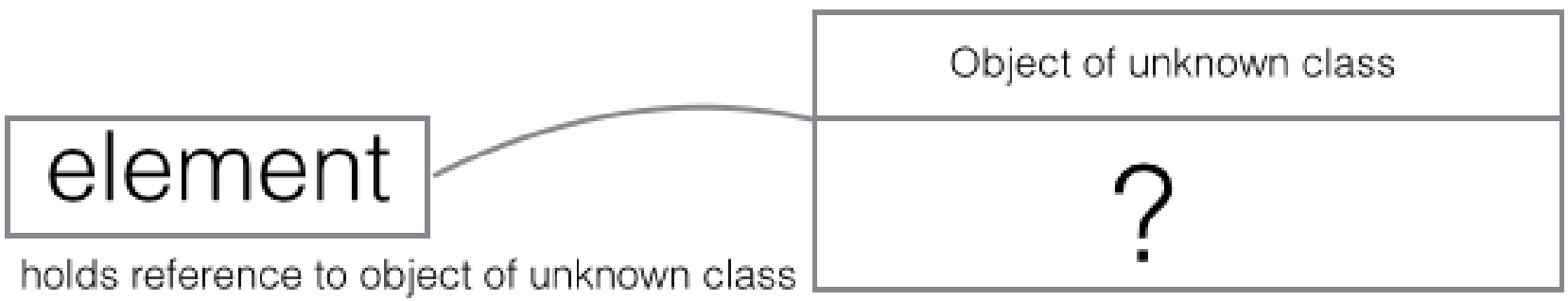
```
/*This is allowed*/
Drawable element = new Tree(...);
/*This is not allowed*/
Drawable element = new Drawable();
```

Java *interface*

Polymorphism

- Here *element* a reference to Drawable variable
- No way to know what class type referenced
- Only know object has method *draw()*

```
Array<Drawable> elements;//elements contains Houses, Trees, Triangles,...
Drawable element = elements.get(i);//specific member of elements index i
```



Methods

Synchronous Processing

When client invokes method:

- Execution proceeds only when method returns
- Referred to as synchronous processing

```
public static void main(String[] args)
{
  int limit = 500;
  int val = 0;
  do {
    textview.doWork();
    val += 1;
  } while (val < limit);
}
```

Methods

Example asynchronous Processing

When client invokes method:

- Invoked method kicks off task . . .
- . . . and immediately returns
- Client continues doing other things
- When task complete client somehow advised
- Referred to as asynchronous processing

Listener

Verbose demonstration

Create a simple interface **TextWatcher**

```
public interface TextWatcher
{
  void onTextChanged(String changedtext);
}
```

Listener

Verbose demonstration

Create a class **Callback** that implements the interface

```
public class Callback implements TextWatcher
{
   @Override
   public void onTextChanged(String changedtext)
   {
    System.out.println(changedtext);
   }
}
```

Listener

Verbose demonstration

TextView: Register the listener *TextWatcher*

```
public class TextView
{
  private TextWatcher textwatcher;

  public void addTextChangedListener(TextWatcher textwatcher)
  {
    // Save textwatcher for later use.
    this.textwatcher = textwatcher;
  }
  ...
}
```

Listener

Verbose demonstration

TextView: set predicate & do work

```
public class TextView
{
  ...
  private boolean somethingHappened;

  // Invoking with flag == true sets scene for a callback
  public void setPredicate(boolean flag) {
    somethingHappened = flag;
  }
  // This method will be invoked repeatedly in an event loop
  public void doWork() {
    if (somethingHappened) { // Check the predicate, set elsewhere.
      // Handle the event by invoking the interface's method.
      textwatcher.onTextChanged("Finally - you called back");
      somethingHappened = false;//reset predicate
    }
  }
}
```

Listener

Verbose demonstration

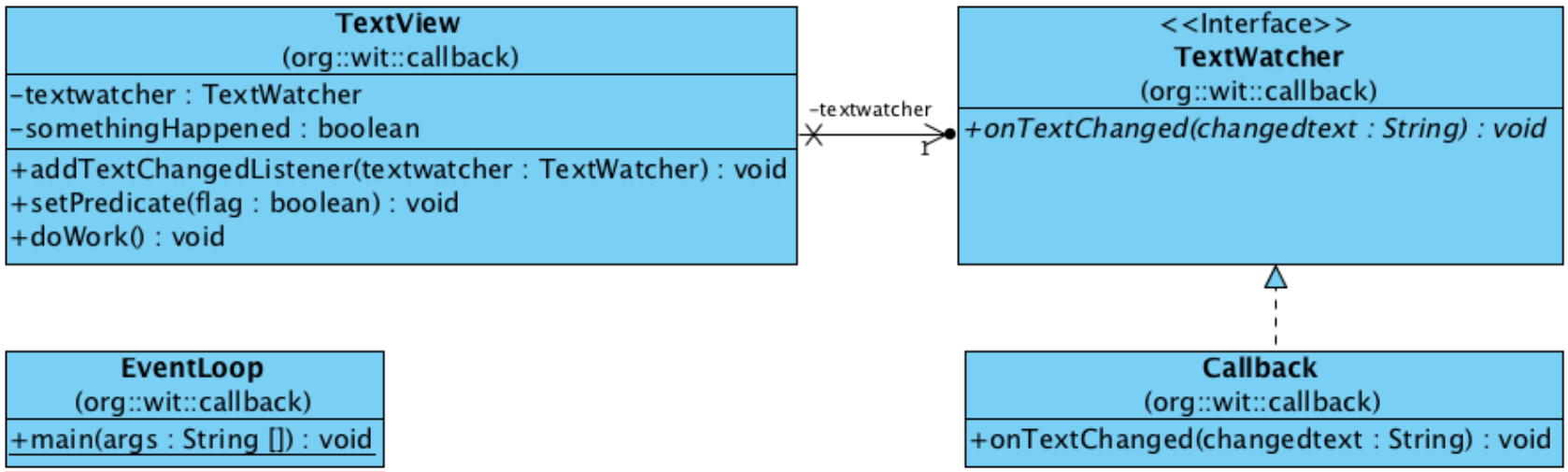
EventLoop class: *main* method

```
//Main method
TextWatcher textwatcher = new Callback();
TextView textview = new TextView();
textview.addTextChangedListener(textwatcher);

int val = 0;
// The simulated event loop
do
{
  if (val % 100 == 0)
  {
    textview.setPredicate(true); // trigger an event
  }
  // invoke repeatedly but trigger event only when predicate true
  textview.doWork();
  val += 1;
} while (val < 500);// we expect 5 events to be triggered
```

Listener

Verbose demonstration



Listener

Using anonymous class

EventLoop class: *main* method: no Callback object

```
TextView textview = new TextView();
// We use an anonymous class instead of the Callback object
textview.addTextChangedListener(new TextWatcher() {
  @Override
  public void onTextChanged(String changedtext) {
    System.out.println(changedtext);
  }
});
int val = 0;
// The simulated event loop
do {
  if (val % 100 == 0) {
    textview.setPredicate(true); // trigger an event
  }
  // invoke repeatedly but trigger event only when predicate true
  textview.doWork();
  val += 1;
} while (val < 500);// we expect 5 events to be triggered
```

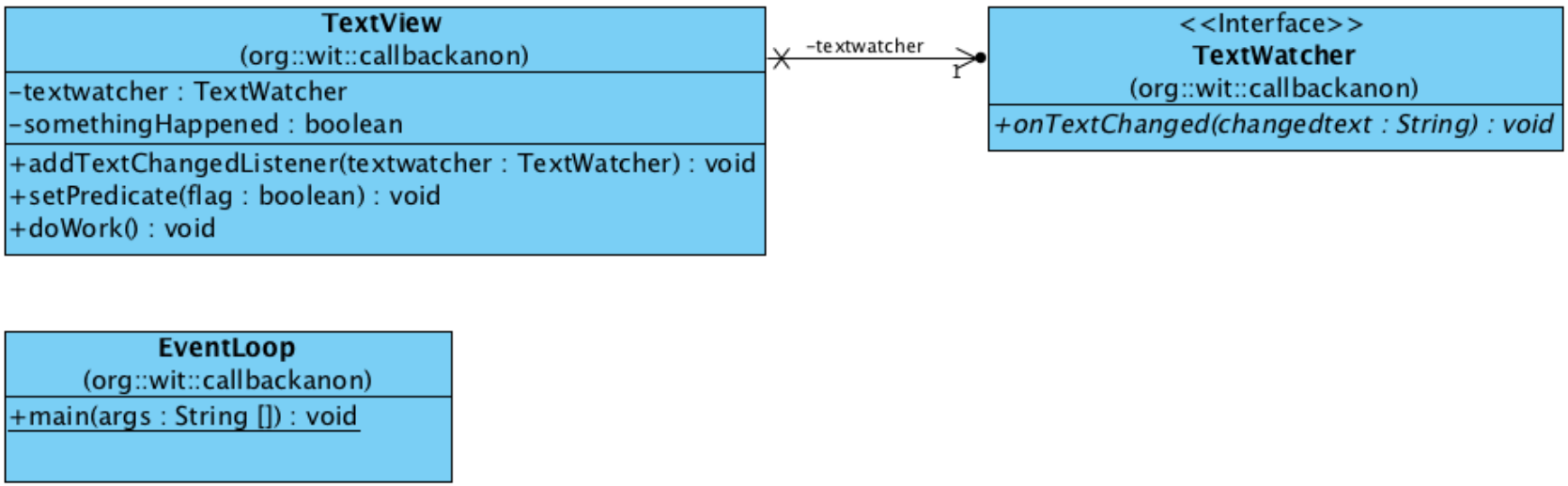
Listener

Using anonymous class

- :Note signature of *onTextChanged*
  - `onTextChangedListener(TextWatcher)`
- But this is how we invoke:
  - `textView.onTextChangedListener(new TextWatcher(){. . .});`
- Not very intuitive: seems to suggest interface being instantiated.

Listener

Using anonymous class



Listener

Avoiding use of anonymous class

EventLoop class: *main* method: not using anonymous class

- Class EventLoop implements TextWatcher
- Use EventLoop **this** as *addTextChangedListener* argument

```
public class EventLoop implements TextWatcher
{

  public void runloop()
  {
    TextView textview = new TextView();

    // EventLoop implements TextWatcher
    // Consequently "this" a legal parameter here
    textview.addTextChangedListener(this);
    //Simulate event loop
    ...
   }
   ...
 }
```

Listener

Avoiding use of anonymous class

EventLoop class: *main* method: not using anonymous class

- Implement interface TextWatcher method in EventLoop

```
public class EventLoop implements TextWatcher
{
  public void runloop() {...}

  @Override
  public void onTextChanged(String changedtext) {
    System.out.println(changedtext);
  }
  public static void main(String[] args)
  {
    EventLoop obj = new EventLoop();
    obj.runloop();
  }
}
```

Listener

Avoiding use of anonymous class