Abstract classes

Abstract classes
Overriding methods
Super
Abstract methods

Abstract classes

Doing the following makes sense:

```
Button okButton = new Button();
```

Doing the following does not:

```
GuiControl mySpecialControl = new GuiControl();
```

What's odd about the second one?

Abstract classes

Basically, **GuiControl** is an **abstract** concept. What does a pure instance of **GuiControl** look like?

A button has a certain look that can be rendered, as well as any other **concrete** control.

We don’t want to be able to create instances of **abstract** concepts like **GuiControl**.

Abstract classes

This can be achieved by marking the class as **abstract**

```
public abstract class GuiControl {
    private int positionX;
    private int positionY;

    public int getPositionX() {
        return positionX;
    }

    public void setPositionX(int positionX) {
        this.positionX = positionX;
    }

    // Other methods omitted
}
```

We call this an **abstract class**

Abstract classes

Trying to instantiate the class won't work as **abstract classes** can't be instantiated.

```
GuiControl mySpecialControl = new GuiControl();
```

```
'GuiControl' is abstract; cannot be instantiated
```

Now, we're only using the **GuiControl** class as a superclass and to support polymorphism.

Overriding methods

Overriding methods (1)

We have the following method in our control superclass:

```
public class GuiControl {

    /**
     * Describes the current position.
     * @return The current position */
    public String describe() {
        return String.format("Position: %1$d %2$d",
                             positionX, positionY);
    }
}
```

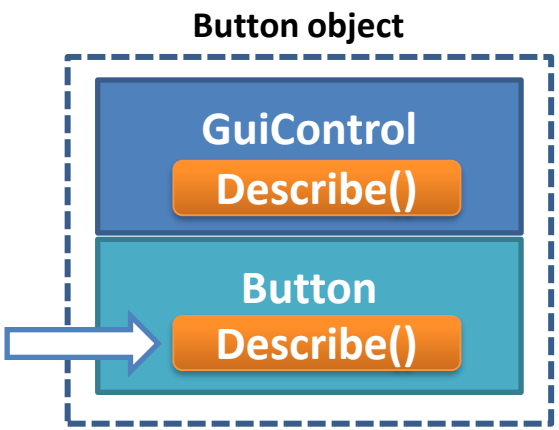
Having this method in the superclass makes sure that an instance of **GuiControl** (such as a **Button** instance) can always call **describe()**.

Overriding methods (2)

While it's fine for a **Button** to describe itself with its coordinates, it would be nice if we could let a **TextBox** object describe its location as well as its **current content**.

The solution comes with **overriding** them in the subclass.

Replace **Describe** in the superclass with one in the subclass.



Overriding methods (3)

To **override** the method in a subclass, add a method with the same name and annotate it with **@Override**.

```
public class TextBox extends GuiControl {
    private String text;

    // Other code omitted

    @Override
    public String describe() {
        return super.describe() + String.format("Content: %s",
                                                 text);
    }
}
```

The **@Override** annotation is optional but good practice.

super is used to call methods in the superclass.

Overriding a method means giving it a **new implementation** in an subclass.

Overriding methods (4)

Now we have:

```
public abstract class GuiControl {
    // Other code omitted

    public String describe() {
        return String.format("Position: %1$d %2$d",
                             positionX, positionY);
    }
}



public class TextBox extends GuiControl {
    private String text;

    // Other code omitted

    @Override
    public String describe() {
        return super.describe() + String.format("Content: %s",
                                                text);
    }
}
```

Using the **describe** method

Here we'll create both a **Button** and a **TextBox** and call the **describe()** method.

```
public static void main(String[] args) {
    Button myButton = new Button();
    myButton.setPositionX(50);
    myButton.setPositionY(120);

    TextBox nameField = new TextBox();
    nameField.setPositionX(10);
    nameField.setPositionY(30);
    nameField.setText("Edument");

    // Button will use the superclass version:
    System.out.println(myButton.describe());

    // TextBox will use the overridden version:
    System.out.println(nameField.describe());
}
```

The implementation of **describe()** will depend on the control type.

Output

If we run the code:

```
public static void main(String[] args) {
    // Code omitted

    System.out.println(myButton.describe());
    System.out.println(nameField.describe());
}
```

We would end up with the following output:

```
Position: 50 120
Position: 10 30 Content: Edument
```

Final methods

Sometimes we don't want to let a method be overridden.

```
public abstract class GuiControl {
    // Other code omitted

   /**
    * Describes the current position.
    * @return The current position
    */
    public final String describe() {
        return String.format("Position: %1$d %2$d",
                             positionX, positionY);
    }
}
```

Now **describe** can NOT be overridden by inherited classes.

In this case, we declare the method as **final**, which prevents it being overridden by subclasses.

Final methods

final also work on a class level:

```
public final class Button extends GuiControl {
    // Text label to display on screen
    private String buttonLabel;

    // Getters/Setters omitted

    public void onClick() {
        // Handle a click
    }
}


public class SuperButton extends Button {

    //Code...

}
```

Cannot inherit from final 'com.company.Button'

In this case, we declare the class as **final**, which prevents it being extended/inherited by subclasses.

Calling super

Calling **super** (1)

What if the superclass and subclass has a **constructor**?

```
public class GuiControl {

    public GuiControl()
    {
        System.out.println("GuiControl constructor");
    }
}

public class Button extends GuiControl {
    // Text label to display on screen
    public Button()
    {
        System.out.println("Button constructor");
    }
}
```

What will the output be if we create an instance?

```
Button okButton1 = new Button();
```

```
GuiControl constructor
Button constructor
```

Calling **super** (2)

What if the superclass has a **constructor** with parameters?

```
public abstract class GuiControl extends Object {
    private String controlName;
    private int positionX;
    private int positionY;

    public GuiControl(String controlName) {
        this.controlName = controlName;
    }

    public String describe() {
        return String.format("Position: %1$d %2$d",
                positionX, positionY);
    }
}

public class Button extends GuiControl {
    // Text label to display on screen
    public Button()
    {
        System.out.println("Button constructor");
    }
}
```

The Subclass must call this constructor.

This code will no longer compile!

Calling **super** (3)

We have to add a call to **super()** in the constructor of the subclass.

super() calls the constructor in the class we've extended.

```
public class TextBox extends GuiControl {
    // Code omitted

    public TextBox() {
        super("TextBox");
    }

    @Override
    public String describe() {
        return super.describe() + String.format("Content: %s",
                                                text);
    }
}
```

This constructor will call the superclass constructor with the argument "**TextBox**".

super points to the immediate parent class

Abstract methods

Abstract methods

As we've seen, it sometimes makes sense being unable to instantiate a class (by making it **abstract**), as with **GuiControl**.

In a similar way, sometimes it doesn't make sense to provide an implementation to a method. In that case, you can make methods **abstract** as well.

An abstract method has no body and **must** be overridden in its subclasses.

Abstract method: Example (1)

Every control needs to be drawn (or rendered).

We could provide a new method called **render** in the superclass to do this.

```
public abstract class GuiControl {
    // Code omitted

    public void render() {
        // Render the control
    }
}
```

Abstract method: Example (2)

However, a **GuiControl** is an abstract concept and can't be drawn.

The **concrete** classes (**Button**, **TextBox** and so on) have actual shapes that we could draw.

In other words: we want to **override** the **render()** method in subclasses, but we don't want to give it any implementation at all in the superclass.

We want to force the subclasses to always override it!

Abstract method: Example (3)

Make the **render** method **abstract** by adding the keyword and removing the body entirely:

```
public abstract class GuiControl {
    // Code omitted

    public abstract void render();
}
```

No body { }

Abstract method: Example (4)

When inheriting from a class with **abstract** methods, we **have** to override and provide an implementation.

```
public class TextBox extends GuiControl {
    // Code omitted
    public TextBox() {
        super("TextBox");
    }

    @Override public String describe() {
        return super.describe() + String.format("Content: %s",
                                                text);
    }
}
```

```
Class 'TextBox' must either be declared abstract or
implement abstract method 'render()' in 'GuiControl'
```

Abstract method: Example (5)

Doing so, is just like overriding any method:

```
public class TextBox extends GuiControl {
    // Code omitted

    public TextBox() {
        super("TextBox");
    }

    @Override public String describe() {
        return super.describe() + String.format("Content: %s",
                                                text);
    }

    @Override
    public void render() {

    }
}
```

Summary

- Use **abstract classes** to create common functionality for classes, but where the abstract concept shouldn't be instantiated.
- **Abstract methods** must always be overridden in non-abstract subclasses.
- Only **abstract classes** can contain **abstract methods**.
- Mark field variables/methods as **final** to prevent them being overridden.
- In a subclass, it is good practice to mark with **overrides** with **@Override**.

Exercise 19

Lets do exercise 19