Inheritance

Inheritance

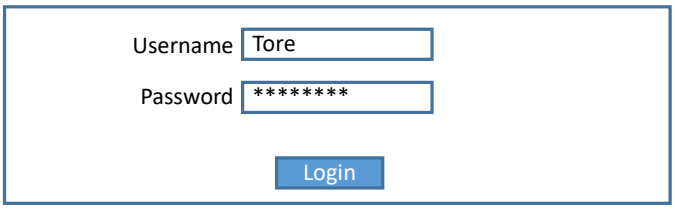
Inheritance adds reusability to code.

Inheritance lets us write a **superclass** (or **base class**) which contains some kind of ”general” functionality.

Other classes, called **subclasses** can **inherit** common behaviour from its superclass and add its own **specialized** functionality to it.

Inheritance: Example (1)

Lets say we want to develop a **G**raphical **U**ser **I**nterface (GUI) system



How would we write the code to describe the GUI above?

Inheritance: Example (2)

The following class describes a **button** for a GUI.

```
public class Button {
    // The button's position
    private int positionX;
    private int positionY;

    // A text label to display on screen
    private String buttonLabel;

    // Getters/Setters omitted

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

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

The result is formatted as a decimal integer

Inheritance: Example (3)

Let's add a **TextBox**, where the user can enter text.

```java
public class TextBox {
    // The text box's position
    private int positionX;
    private int positionY;

    // The text to display in the text box
    private String text;

    // Getters/Setters omitted

    public void onTextChange() {
        // Handle the event
    }

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


Inheritance: Example (4)

We will probably have more than these two GUI controls too.

- Dropdown lists
- Checkboxes
- Scrollbars
- Text labels

.
.
.

Inheritance: Example (5)

Let's compare these classes

Redundant!

```java
public class Button {
    //Button position
    private int positionX;
    private int positionY;

    //Text label to display on screen
    private String buttonLabel;

    // Getters/Setters omitted

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

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

```java
public class TextBox {
    // Text box position
    private int positionX;
    private int positionY;

    // Text in text box
    private String text;

    // Getters/Setters omitted

    public void onTextChange() {
        // Handle the event
    }

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

The classes are very similar. They are both **GUI controls** with a certain set of common behaviour.

Inheritance: Example (6)

However, they are still **different enough** that they should be separate classes.

Similarities:

- The field **position** is common to all of these classes.
- The method **describe()** is also common to them.
- If we want to add another member to all of our GUI controls, we would have to change a lot of classes.

What if we could put all the common things in something that can be shared and reused?

The Super class

Super class (1)

To improve this we will create a **superclass** called **GuiControl** to hold these members.

```
public class GuiControl {
    // Control position
    private int positionX;
    private int positionY;

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

Other common terms for superclass is **base class** or **parent class**

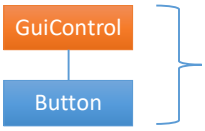
Super class (2)

Every control will **inherit** from this class, meaning we don't have to declare **position** and **describe()** in each and every GUI control class.

Syntax for inheritance:

SubClass **inherits** BaseClass

```
public class Button extends GuiControl
```



Button class will now contain the combined members

Super class (3)

Revised subclasses for the controls.

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

    // Getters/Setters omitted

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

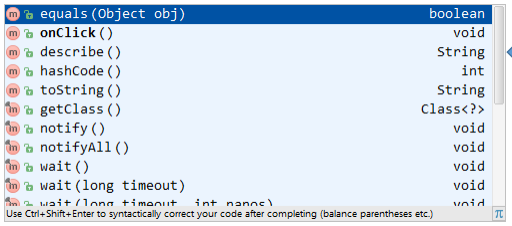
```
public class TextBox extends GuiControl {
    // Text in text box
    private String text;

    // Getters/Setters omitted

    public void onTextChange() {
        // Handle the event
    }
}
```

Note that we have removed the position **fields** and the **describe** method from the subclasses.

Super class (4)

All classes that inherit from **GuiControl** will now contain the members **position** and **describe()**.

```
Button btn = new Button();

btn.
```



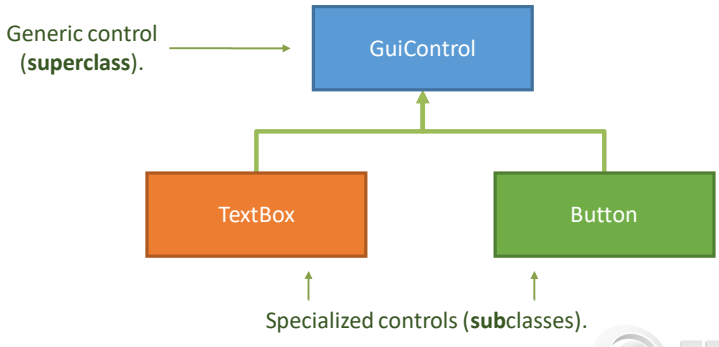
The public Describe Method from the superclass

The **public** members of the superclass are also public members of the subclasses and show up in IntelliSense as any other member.

Class hierarchy

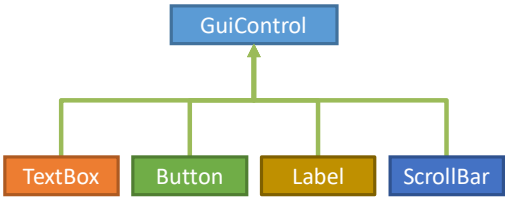
Hierarchy

We could visualize the **inheritance hierarchy** in the following way:



Hierarchy

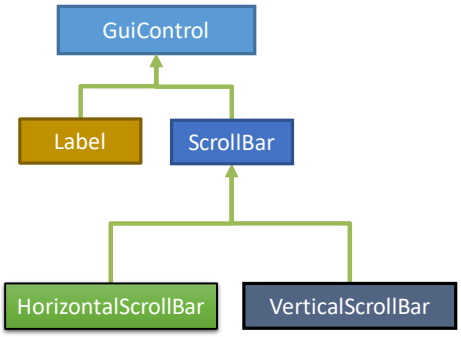
We can add more specialized versions, all inheriting from **GuiControl**.




```
public class Label extends GuiControl {
    // ...
}

public class ScrollBar extends GuiControl {
    // ...
}

public class Button extends GuiControl {
    // ...
}

public class TextBox extends GuiControl {
    // ...
}
```

Hierarchy

Technically, we should be able to use any class as a **superclass**, including classes which themselves are **subclasses**




```
public class ScrollBar extends GuiControl {
    // ...
}

public class VerticalScrollBar extends ScrollBar {
    // ...
}

public class HorizontalScrollBar extends ScrollBar {
    // ...
}
```

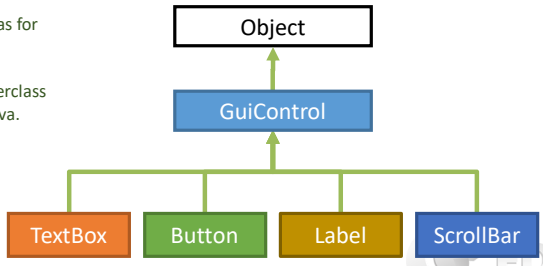

The **object** class

In fact, the **GuiControl** is not the start of the hierarchy.

All classes (and most other types) that you write yourself will inherit from a root type called **Object**.

Object is actually an alias for **java.lang.Object**.

This is the ultimate superclass of all other classes in Java.



The **object** class

For instance, our **GuiControl** class acts as if we were writing code in the following way:

```
public class GuiControl extends Object {
    // Control position
    private int positionX;
    private int positionY;

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

This will happen automatically. You don't have to inherit explicitly from object.

The **object** class

The **Object** keyword is actually an alias for **java.lang.Object**.

This is the ultimate superclass of all other classes in Java.

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

    Object obj;
    obj.
}
```

equals(Object obj)	boolean
hashCode()	int
toString()	String
getClass()	Class<? extends Object>
notify()	void
notifyAll()	void
wait()	void
wait(long timeout)	void
wait(long timeout, int nanos)	void
cast	((SomeType) expr)

Press Ctrl+Period to choose the selected (or first) suggestion and insert a dot afterwards >>

Members provided by Java.lang.Object

Accessibility and inheritance

public
private
protected

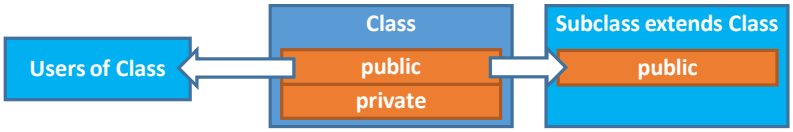
Public and private

public is always **public**.

A public member in a class is accessible to all classes and sub-classes

private means **hidden**.

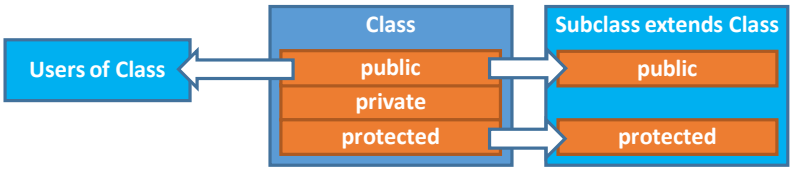
A private member in a class is inaccessible to all classes (including subclasses) except for the class itself.



Protected

protected is somewhere in-between.

A **protected** member in a class is accessible to the class itself and any subclasses, but is not accessible externally.



Summary

When should we use inheritance?

Reusable code

- Changes in the superclass are reflected in all subclasses.

Usable when:

- We want several classes to share some behaviour, and…
- We want every subclass to extend the superclass behaviour.

Use inheritance when you want to take one class with a certain behaviour and **specialize** it.

Applying inheritance

Inheritance is a tool, and tools should be used where applicable.

Applying inheritance

When figuring out your class structure, a good rule of thumb is to reflect on what kind of **relationship** you're working with.

Inheritance describes a **is a** relationship. Button **is a GuiControl**, **VerticalScrollBar is a ScrollBar**.

If what you have is a **has a** relationship, you're probably looking for **composition** and not **inheritance**. A **Car has an** engine, a car **has** doors.

Exercise 17

Let's do exercise 17