

# Inheritance

# Extending Classes

- It's possible to create a class by using another as a starting point.
  - We can take the existing class: add more methods, change what methods do, etc.
- Allows reuse.
  - Can extend a class in several ways, using it for several different purposes.
  - ... based on the same original code.

# Extending Classes

- The class that's being extended is the “parent class” or “superclass”.
  - Contains all behaviour common to its “children”.
- The parent is extended in the “child class” or “subclass”.
  - Additional, specific behaviour added.
  - Often a parent is extended in several ways, creating several children.

# Example

- In the blackjack design, we had players and dealers that were similar.
  - Each had a hand, took a turn, etc.
  - Put this in a superclass `Person`.
- Then, the person class can be extended to create `Player` and `Dealer`.
  - Code/behaviour that differs between the two is added as part of the subclass.

# Inheriting

- Suppose we have a simple class A:

```
class A {  
    private int count;  
    public int method() {  
        return 1;  
    }  
}
```

- ... and want to create a class B that is A with another method added...

# Inheriting

- Now, B can inherit A and add a new method:

```
class B extends A {  
    public int newMethod() {  
        return 2;  
    }  
}
```

- B now has everything from the definitions of A and B.

# Inheriting

- Using A and B:

```
A myA = new A();
```

```
B myB = new B();
```

```
System.out.print(myA.method());
```

```
System.out.print(myB.method());
```

```
System.out.print(myB.newMethod());
```

Originally from A




Only available in B

- Output: 112

- An instance of B also has all members from A.

# Parents and Privacy

- Subclasses **cannot** access private members:

```
class B extends A { ...  
    public void setCount(int count) {  
        this.count = count;   
    }  
}
```

Error: count defined as  
private in A; we aren't in A.

- Just like any other code: it can't access private members.
- Must use getters & setters on instance variables.



# Constructors

- Constructors are **not** inherited, but can be added the same way:

```
class A { ...  
    public A() {  
        count = 1;  
    }  
}
```

Still named after the class they're defined in.

```
class B extends A { ...  
    public B(int n) {  
        setCount(n);  
    }  
}
```

count is private: must use setter to assign.

# Overriding Methods

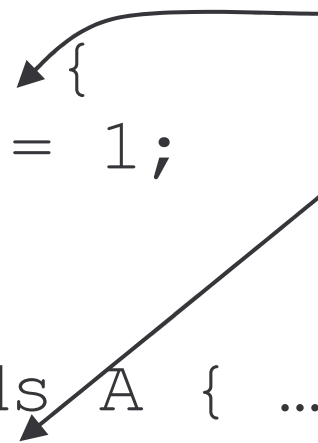
- If you want to change the behaviour of a method, it can be redefined in the subclass
  - “overriding”
  - Any instances of the child class will use this implementation, not the parent’s.
- Useful if a method’s behaviour needs to be different for the specific subclass
  - e.g. extra variables updated, different output, ...

# Example

- Works for constructors too:

```
class A { ...  
    public A() {  
        count = 1;  
    }  
}  
class B extends A { ...  
    public B() {  
        setCount(6);  
    }  
}
```

Same argument signature, so the original is overridden.



# Accessing the Superclass

- If you need to access something from the superclass, the `super` reference gives access.
  - Particularly useful for calling overridden methods.

- Common usage:

```
public void method(int x) { // Overrides
    ...                    // Extra stuff here.
    super.method(x); // Run the superclass
                       // method too.
}
```

# Built-In Classes

- It's also possible to extend classes from the standard library.
  - Even if you don't know the original implementation, you can still add new methods.
  - eg. list with metadata (`ArrayList` with extra instance variables); `Scanner` that can also read in a custom type.

# The Object Class

- The built-in class `Object` is the “root” of the class hierarchy.
  - A class that doesn’t explicitly “extend” anything has `Object` as its superclass.
  - i.e. these are equivalent:

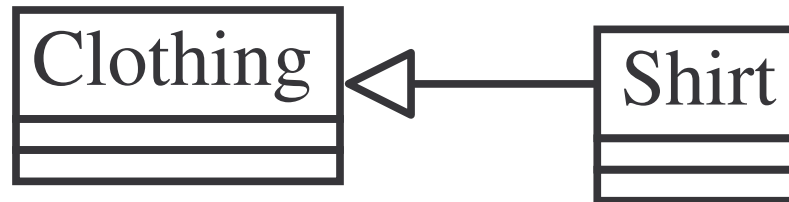
```
class Foo {...}
class Foo extends Object {...}
```
- `Object` contains some methods (`toString`, `equals`, ...) that are used unless overridden.

# When to Inherit

- Inheritance is a very powerful tool in OO design.
  - Easy reuse of code; works naturally with problems that have a hierarchy of objects.
- Don't confuse inheritance with aggregation.
  - A should inherit B only if “A is a B” or “A is a more specific version of B” or “A can be substituted for B”
  - ... not just because you need to use the parts of B.

# Inheritance in UML

- The inherited class is indicated with a solid line and open arrow:





# Example: Shapes

# Shapes

- Suppose we want to represent a collection of shapes (in a drawing program).
  - square, rectangle, circle (maybe other later).
- All of these have some things in common:
  - `position`: the  $xy$ -coordinate of the shape
  - Getters & setters for position.
  - `translate(x, y)`: move the shape by this much
- Create a class `Shape` that implements these.

# Subclasses

- Each of these will inherit Shape.
- Rectangle:
  - Add instance variables `width`, `height`.
- Square:
  - Inherit Rectangle and override setters to ensure `width==height`.
- Circle:
  - Add `radius`, interpret the `position` as the centre.

# Code Sketch

- The class definitions:

```
class Shape { ... }
```

```
class Rectangle extends Shape { ... }
```

```
class Square extends Rectangle { ... }
```

```
class Circle extends Shape { ... }
```

- A square is a specific kind of rectangle, so we should have the inheritance this way.

- not `Rectangle extends Square`

# (Partial) UML Diagram

