# INFO1113 / COMP9003
# Object-Oriented Programming

## Lecture 5

THE UNIVERSITY OF SYDNEY

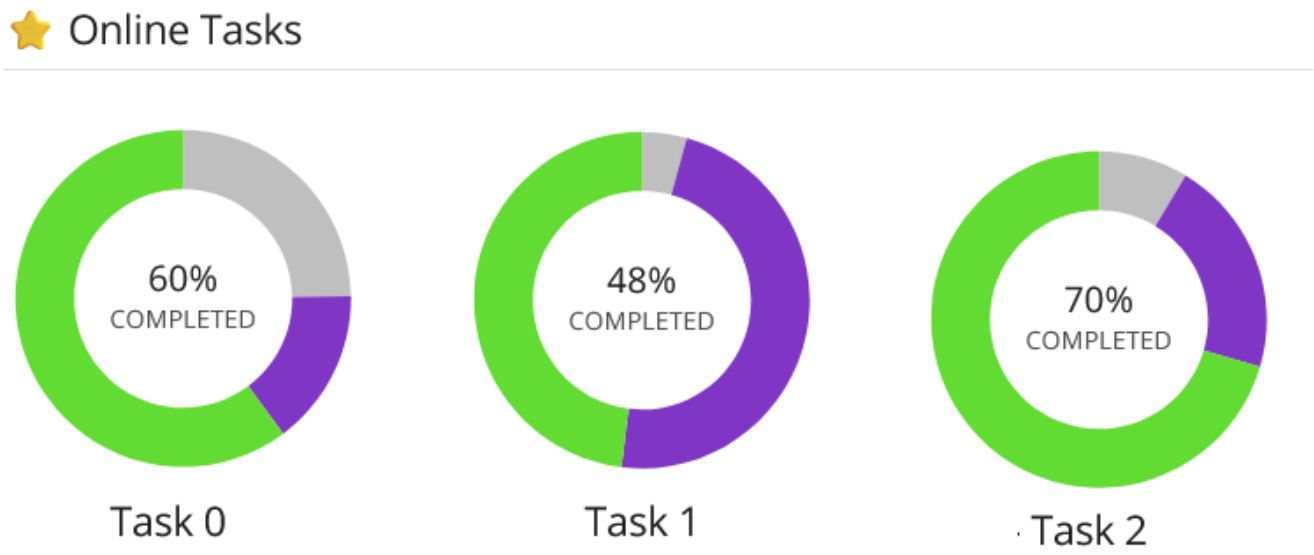
Contents developed by Tyson Thomas

# Acknowledgement of Country

*I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.*

*I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.*

# Always try to be in the green zone!



# Copyright Warning

**COMMONWEALTH OF AUSTRALIA**

**Copyright Regulations 1969**

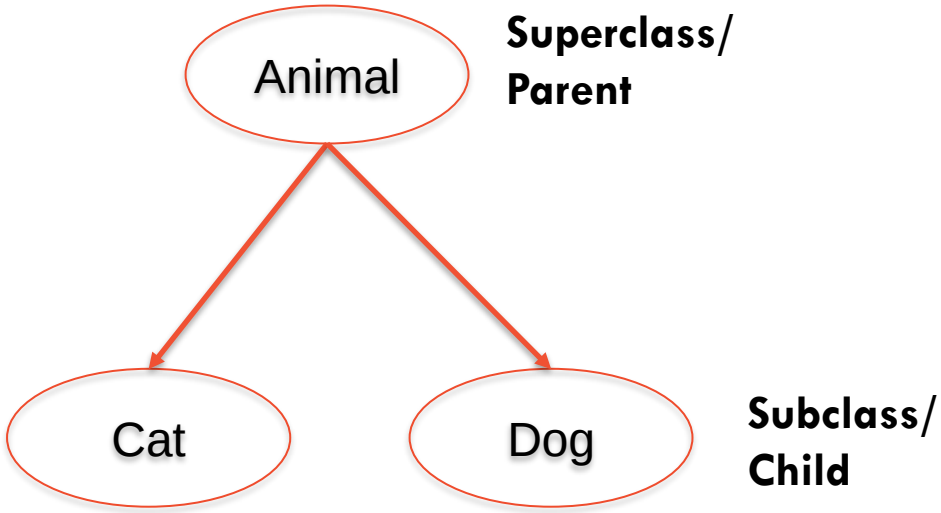
**WARNING**

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

**Do not remove this notice.**

# Topics: Part A




- Inheritance basics
- Encapsulation
- Programming Inheritance
- Modelling an **is-a** relationship and UML

# Inheritance

**Inheritance** is a significant concept of **OOP**. Allowing reusability and changes to inherited methods between different types in a **hierarchy**.

**What does inheritance offer?**

- Attribute and method reusability
- Defining sub-class methods
- Overriding inherited methods
- Type information

# Inheritance

**How does it work?**

We will be introducing a new keyword today called **extends**, this keyword allows the class to inherit from another class.

**Syntax:**

[public] class ClassName **extends** SuperClassName

- Class definition, we specify the access modifier
- **ClassName** (What you are going to name the class)
- We are inheriting from the following class. It is seen as an **extension** of the super class.
- The class we are inheriting from. It will inherit any **protected** or **public** methods or attributes

Refer to Chapter 8.1, page 624-629 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# How it looks

Part of our class declaration line allows for us to define what class we want to **extend** from

```
public class Dog extends Animal
```

Once defined, **Dog** type can also be used as a **Animal** type as it is just an extension of such type.

Refer to Chapter 8.1, page 624-629 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# Encapsulation

We have used the **public** and **private** access modifier but we will now use the **protected** access modifier.

What does **protected** mean?

Like **private** it will not be accessible to other classes but now with the exception **inherited classes.**

- Is only accessible within the class
- Attributes and methods will be accessible by all subclass

Refer to Chapter 8.1, page 632 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;


   public double volume() {
      return height*width*depth;
   }

}
```

```
public class GlassBottle extends Bottle {


   private boolean shattered = false;


   public boolean isBroken() {
      return shattered;
   }
}
```

# Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public double volume() {
        return height*width*depth;
    }

}
```

**Subclass** will have access to any **protected** and **public** methods.

```
public class GlassBottle extends Bottle {

    private boolean shattered = false;

    public boolean isBroken() {
        return shattered;
    }
}
```

# Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;


   public double volume() {
      return height*width*depth;
   }

}
```

**Protected** like **private** but allows subclass to inherit the property.

```
public class GlassBottle extends Bottle {

   private boolean shattered = false;


   public boolean isBroken() {
      return shattered;
   }
}
```

# Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;


   public double volume() {
      return height*width*depth;
   }

}
```

```
public class GlassBottle extends Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;
   private boolean shattered = false;


   public boolean isBroken() {
      return shattered;
   }
}
```

All properties from the **super** class are **inherited** by the **subclass.** As if they were defined in the class itself.

# Inheritance

So let's take a look how inheritance works between two classes.

```
public class Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;


   public double volume() {
      return height*width*depth;
   }

}
```

```
public class GlassBottle extends Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;
   private boolean shattered = false;

   public void shatter() {
      System.out.println("We lost " + litresFilled + "Litres");
      litresFilled = 0;
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

Able to refer to the attributes within the **subtypes** own methods.

## What about constructors?

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle() {

   }

   public double volume() {
      return height*width*depth;
   }

}
```

```
public class GlassBottle extends Bottle {


   private boolean shattered = false;

   public void shatter() {
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {

  protected String name;
  protected double width;
  protected double height;
  protected double depth;
  protected double litresFilled;

  public Bottle() {
```

```
public class GlassBottle extends Bottle {

  private boolean shattered = false;

  public void shatter() {
    shattered = true;
```

By default, when a **subclass** object is created, it will refer to the **super** class's constructor.

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

  GlassBottle b = new GlassBottle();
  System.out.println(b.isBroken());
  System.out.println(b.name);
}
```

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {

   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle() {

   }

   public double volume() {
      return height*width*depth;
   }

}
```

```
public class GlassBottle extends Bottle {

   private boolean shattered = false;

   public void shatter() {
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

However! Nothing was initialised, so all we get are default values

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle() {
      this.name = "Basic Bottle";
      this.width = 10.0;
      this.height = 10.0;
      this.depth = 10.0;
      this.litresFilled = 0;
   }

   public double volume() {
      return height*width*depth;
   }
}
```

Providing some values we can inspect the previous code segment

```
public class GlassBottle extends Bottle {

   private boolean shattered = false;

   public void shatter() {
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10.0;
        this.height = 10.0;
```

```
public class GlassBottle extends Bottle {

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
```

By default, when a **subclass** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.isBroken());
    System.out.println(b.name);
}
```

```
> java MyProgram
false
Basic Bottle
<program end>
```

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle() {
      this.name = "Basic Bottle";
      this.width = 10.0;
      this.height = 10.0;
      this.depth = 10.0;
      this.litresFilled = 0;
   }


   public double volume() {
      return height*width*depth;
   }
}
```

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      this.name = "Glass Bottle";
   }

   private boolean shattered = false;

   public void shatter() {
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

What if we were to define a constructor in the subclass?

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle() {
      this.name = "Basic Bottle";
      this.width = 10.0;
      this.height = 10.0;
```

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      this.name = "Glass Bottle";
   }

   private boolean shattered = false;

   public void shatter() {
```

By default, when a **subclass** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
   GlassBottle b = new GlassBottle();
   System.out.println(b.volume());
   System.out.println(b.name);
}
```

```
> java MyProgram
1000.0
Glass Bottle
<program end>
```

We can see that we called the **GlassBottle** constructor and it set the **name** to **Glass Bottle**.

# Inheritance

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle() {
      this.name = "Basic Bottle";
      this.width = 10.0;
      this.height = 10.0;
```

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      this.name = "Glass Bottle";
   }

   private boolean shattered = false;

   public void shatter() {
```

By default, when a **subclass** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
   GlassBottle b = new GlassBottle();
   System.out.println(b.volume());
   System.out.println(b.name);
}
```

```
> java MyProgram
1000.0
Glass Bottle
<program end>
```

Hang on! If we called GlassBottle() how is volume returning 1000.0?

## Let’s try something

# Inheritance

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle(String name, double width,
      double height, double depth) {
      this.name = name;
      this.width = width;
      this.height = height;
      this.depth = depth;
      this.litresFilled = 0.0;
   }


   public double volume() {
      return height*width*depth;
   }
}
```

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      this.name = "Glass Bottle";
   }

   private boolean shattered = false;

   public void shatter() {
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

# Inheritance

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle(String name, double width,
      double height, double depth) {
      this.name = name;
      this.width = width;
      this.height = height;
      this.depth = depth;
      this.litresFilled = 0.0;
   }

   public double volume() {
      return height*width*depth;
   }
}
```

What if we were to add a constructor with parameters?

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      this.name = "Glass Bottle";
   }

   private boolean shattered = false;

   public void shatter() {
      shattered = true;
   }

   public boolean isBroken() {
      return shattered;
   }
}
```

# Inheritance

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle(String name, double width,
      double height, double depth) {
      this.name = name;
      this.width = width;
```

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      this.name = "Glass Bottle";
   }

   private boolean shattered = false;

   public void shatter() {
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods.

```
public static void main(String[] args) {
   GlassBottle b = new GlassBottle();
   System.out.println(b.volume());
   System.out.println(b.name);
}
```

How would the GlassBottle constructor be able to invoke the super constructor?

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class Bottle cannot be applied to given
types;
            public GlassBottle() {
                   ^
 required: String,double,double,double
 found: no arguments
 reason: actual and formal argument lists differ in length
1 error
```

# Inheritance

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

```
public class GlassBottle extends Bottle {

    public GlassBottle() {
        super("", 0.0, 0.0, 0.0);
        this.name = "Glass Bottle";

    }

    private boolean shattered = false;
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```
public static void main(String[] args) {
    GlassBottle b =
    System.out.pr
    System.out.println(b.name);
}
```

We are able to use the **super** keyword to invoke the **parent** constructor.

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class Bottle cannot be applied to given
types;
            public GlassBottle() {
                   ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

# Inheritance

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle(String name, double width,
      double height, double depth) {
      this.name = name;
      this.width = width;
```

Refers to **Bottle** constructor

```
public class GlassBottle extends Bottle {

   public GlassBottle() {
      super("", 0.0, 0.0, 0.0);
      this.name = "Glass Bottle";

   }

   private boolean shattered = false;
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```
public static void main(String[] args) {
   GlassBottle b =
   System.out.pr
   System.out.println(b.name);
}
```

We are able to use the **super** keyword to invoke the **parent** constructor.

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class Bottle cannot be applied to given
types;
            public GlassBottle() {
                   ^
 required: String,double,double,double
 found: no arguments
 reason: actual and formal argument lists differ in length
1 error
```

# Inheritance

We could match the constructor of the parent type.

```
public class Bottle {
   protected String name;
   protected double width;
   protected double height;
   protected double depth;
   protected double litresFilled;

   public Bottle(String name, double width,
      double height, double depth) {
      this.name = name;
      this.width = width;
```

```
public class GlassBottle extends Bottle {

   public GlassBottle(String name, double
      width,double height,double depth){
      super(name, width, height, depth);

   }

   private boolean shattered = false;
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```
public static void main(String[] args) {
   GlassBottle b = new GlassBottle();
   System.out.println(b.volume());
   System.out.println(b.name);
}
```

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class Bottle cannot be applied to given
types;
            public GlassBottle() {
                   ^
 required: String,double,double,double
 found: no arguments
 reason: actual and formal argument lists differ in length
1 error
```

# Demonstration

# Relationship

There are two types of relationships we will look at when it comes to inheritance.

- **Is-a** relationship (Extension)
- **Has-a** relationship (Composition)

In regards to class inheritance we are considering the **Is-a** relationship how a class is an **extension** of another class but is also the other class.

# Relationship

We have to be very **certain** with inheritance that any class that inherits from another **is a** type of that class. There should be clear reasoning that the types satisfy the relationship.

There needs to be clear reasoning to extending the super class.

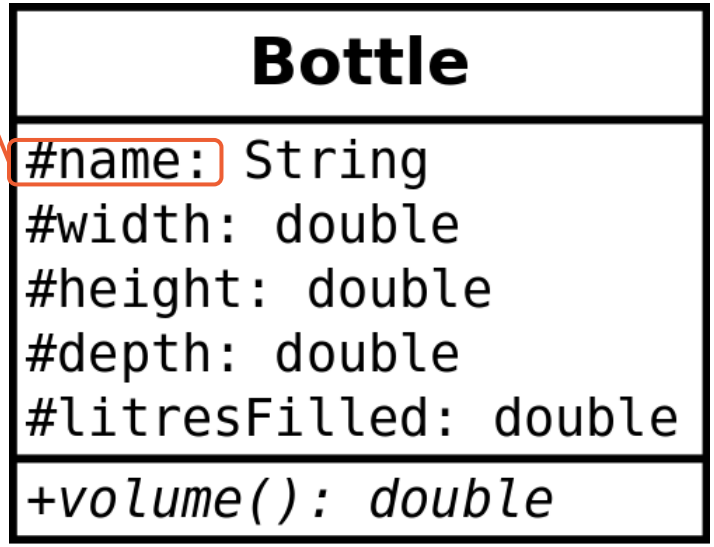
Some instances where it makes sense:

- Super class is **Cat** and subclasses are **Panther, Lion, Tiger**
- Super class is **Controller** and subclasses are **Gamepad, Joystick, Powerglove**
- Super class is **Media** and subclasses are **DVD, Book, Image**

# UML Generalization

Let's examine the following UML Diagram.

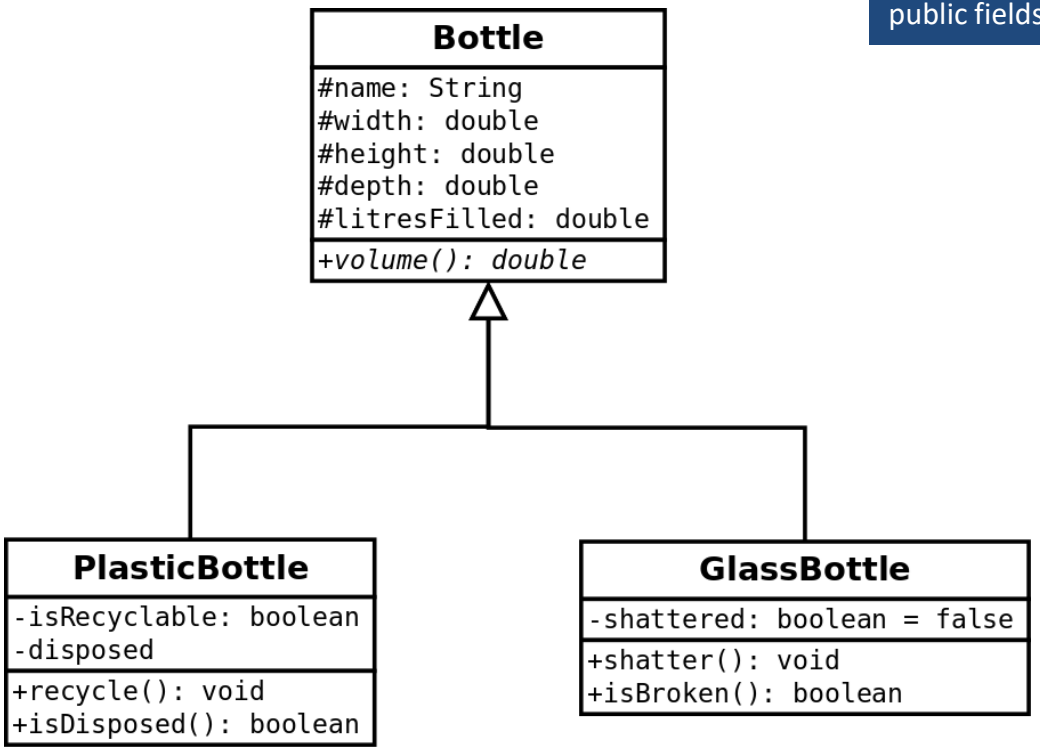
Protected is defined using the # symbol and will be a variable that is inherited.



# UML Generalization

Let's examine the following UML Diagram.

When other classes inherit from the superclass they will get the protected and public fields

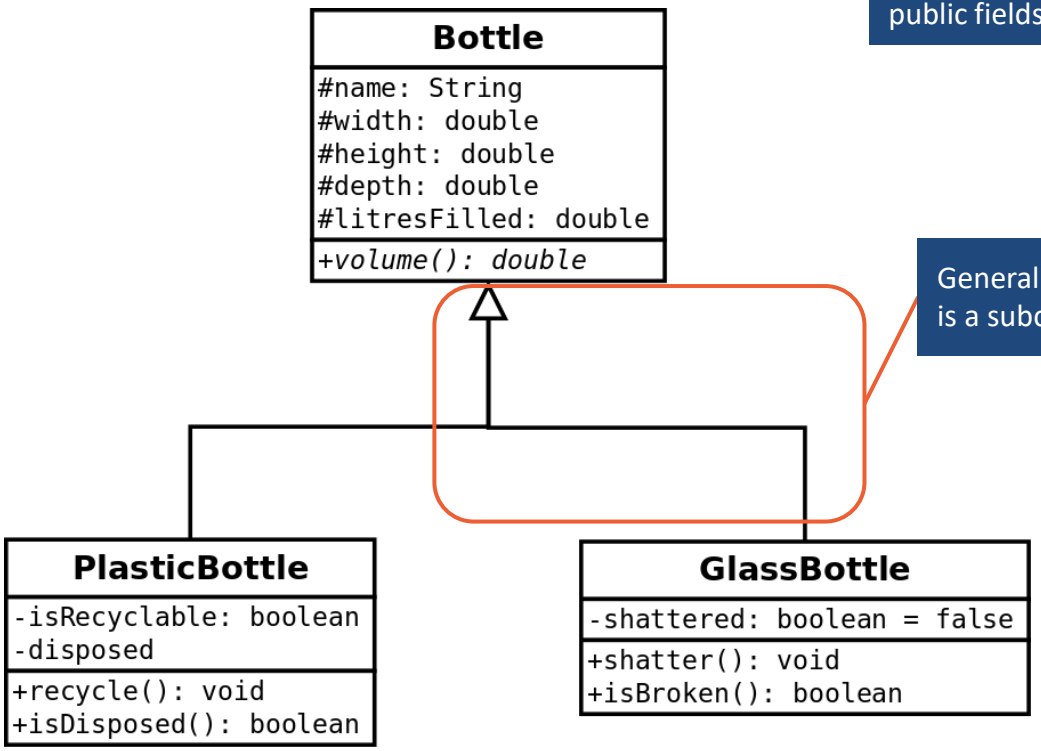



Refer to Chapter 8.1, page 635 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# UML Generalization

Let's examine the following UML Diagram.

When other classes inherit from the superclass they will get the protected and public fields




Generalization link, shows that **GlassBottle** is a subclass of **Bottle.**

Refer to Chapter 8.1, page 635 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

# Super class and subclass

Some other factors to consider:

- Superclass does not know about its subclasses
- Subclasses cannot be constructed using a superclass constructor

  Subclass a = new Superclass(); X

  Superclass a = new Subclass(); ✓
- You **cannot** use subclass properties through a superclass binding.
- **Private** is not inherited, only **protected** and **public**
- Ensure when you use inheritance you are certain it will satisfy an **is-a** relationship
- You can only inherit from **1 class**.
- Within **UML,** inheritance is shown as a **Generalization**.

# Let’s take a break!

THE UNIVERSITY OF SYDNEY

# Topics: Part B

- **Method Overloading**
- **Constructor Overloading**
- **Organizing your application**

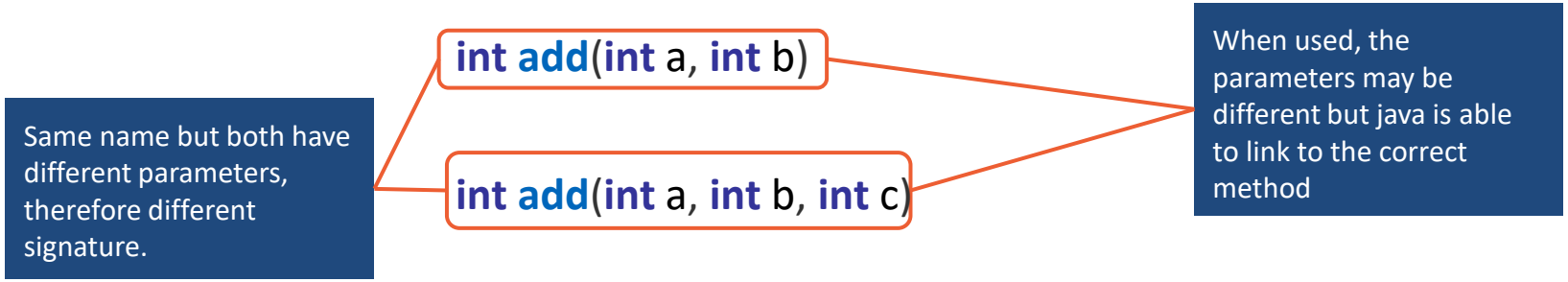
# Overloading

Firstly! What is **overloading?**

In regards to **Java** we are able to use the same method name but with different method **signature**.

Simply:

We are able to define a **method** such as **add** and have a version that accepts two integers and another version that accepts three integers.



# Where it is invalid

We are unable to apply overloading if we have a different **return type** between the methods. The return type is not part of the method signature.

For example:

```
float[] crossProduct(float[] a, float[] b)

int[] crossProduct(float[] a, float[] b)
```

Even though float[] and int[] are specified here, the compile cannot specify which method it will call.

## What about some ambiguous scenarios?

# Ambiguous scenario

So let's consider the following method calls using the two methods and assume that they are correct.

```
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

int[] y = crossProduct(null, null);
```

Which method could it be calling?

The compiler would be unable to determine exactly what method is trying to be called and will throw an error.

```
> javac OverloadTest.java
OverloadTest.java:15: error: reference to crossProduct is ambiguous
        int[] o = crossProduct(null, null);
        ^
  both method crossProduct(float[],float[]) in OverloadTest and method
crossProduct(int[],int[]) in OverloadTest match
1 error
```

# Ambiguous scenario

So let's consider the following method calls using the two methods and assume that they are correct.

```
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct((int[])null, (int[])null);
```

By casting the reference to a certain type, the compiler can deduce what method to call

```
int[] y = crossProduct((float[])null, (float[])null);
```

By casting float[] on the null references we can see it infer the method with floats as arguments

## So, let's demo this!

# Constructor Overloading

We can observe the same overloading concept applied to constructors. This can be applied to both overloaded constructors within the same class as well as super constructors.

We are able to also utilise certain constructors for other constructors if we have already defined that behaviour.

# Java IO

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!

***Constructor Summary***

**Constructors**

| Constructor and Description |
| --- |
| `Scanner(File source)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(File source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(InputStream source)`<br>Constructs a new Scanner that produces values scanned from the specified input stream |
| `Scanner(InputStream source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified input stream |
| `Scanner(Path source)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(Path source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(Readable source)`<br>Constructs a new Scanner that produces values scanned from the specified source. |
| `Scanner(ReadableByteChannel source)`<br>Constructs a new Scanner that produces values scanned from the specified channel. |
| `Scanner(ReadableByteChannel source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified channel. |
| `Scanner(String source)`<br>Constructs a new Scanner that produces values scanned from the specified string. |



Constructor Summary, Oracle (https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)

# Java IO

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!

***Constructor Summary***

**Constructors**

| Constructor and Description |
| --- |
| `Scanner(File source)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(File source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(InputStream source)`<br>Constructs a new Scanner that produces values scanned from the specified input stream. |
| `Scanner(InputStream source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified input stream. |
| `Scanner(Path source)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(Path source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified file. |
| `Scanner(Readable source)`<br>Constructs a new Scanner that produces values scanned from the specified source. |
| `Scanner(ReadableByteChannel source)`<br>Constructs a new Scanner that produces values scanned from the specified channel. |
| `Scanner(ReadableByteChannel source, String charsetName)`<br>Constructs a new Scanner that produces values scanned from the specified channel. |
| `Scanner(String source)`<br>Constructs a new Scanner that produces values scanned from the specified string. |

We can see two different methods we have been using for **Files** and the other for **Standard Input**.



Constructor Summary, Oracle (https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)

# Constructor Overloading

Let's take a look at the following class

```
public class Person {

   private static int DEFAULT_AGE = 21;

   private String name;
   private int age;

   public Person() {
      name = "Jeff";
      age = DEFAULT_AGE;
   }

   public Person(String name) {
      this.name = name;
      this.age = DEFAULT_AGE;
   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

# Constructor Overloading

Let's take a look at the following class

```
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

We can see that there are 3 different constructors. Within our own code we choose to call anyone, in fact we have already been doing this!

# Constructor Overloading

Let's take a look at the following class

```
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }
```

We can see that there are 3 different constructors. Within our own code we

```
public static void main(String[] args) {
    Person p1 = new Person(); //Jeff the default person!
    Person p2 = new Person("Janice");
    Person p3 = new Person("Dave", 32);
}
```

Since each constructor has a unique signature, we are able to utilise specific constructors by satisfying the correct types.

# this keyword

The **this** keyword can play an important role in regards to constructors. It allows us to refer to the constructor within the context of a class.

In particular, we can reduce the amount of code we write by reusing a constructor.

# Constructor Overloading

How could we use the **this** keyword in this example?

```
public class Person {

   private static int DEFAULT_AGE = 21;
   private String name;
   private int age;

   public Person() {
      name = "Jeff";
      age = DEFAULT_AGE;
   }

   public Person(String name) {
      this.name = name;
      this.age = DEFAULT_AGE;
   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

```
public class Person {

   private static int DEFAULT_AGE = 21;
   private String name;
   private int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

# Constructor Overloading

How could we use the **this** keyword in this example?

```
public class Person {

   private static int DEFAULT_AGE = 21;
   private String name;
   private int age;

   public Person() {
      name = "Jeff";
      age = DEFAULT_AGE;
   }

   public Person(String name) {
      this.name = name;
      this.age = DEFAULT_AGE;
   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

```
public class Person {

   private static int DEFAULT_AGE = 21;
   private String name;
   private int age;

   public Person() {
      this("Jeff", DEFAULT_AGE);
   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

# Constructor Overloading

How could we use the **this** keyword in this example?

```
public class Person {

   private static int DEFAULT_AGE = 21;
   private String name;
   private int age;

   public Person() {
      name = "Jeff";
      age = DEFAULT_AGE;
   }

   public Person(String name) {
      this.name = name;
      this.age = DEFAULT_AGE;
   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

By using the **this** keyword, we are able to eliminate few lines from the other constructors by using the last one.

```
public class Person {

   private static int DEFAULT_AGE = 21;
   private String name;
   private int age;

   public Person() {
      this("Jeff", DEFAULT_AGE);
   }

   public Person(String name) {
      this(name, DEFAULT_AGE);
   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

# Demonstration

# Organising your application

# Packages

Java defines a package keyword which will outline to the class which part of the package it resides in. It will self verify on compilation if it exists within the package.

**Syntax:**

**package <identifier>[.<nested ident>[...]]**

# Packages

Java defines a package keyword which will outline to the class which part of the package it resides in. It will self verify on compilation if it exists within the package.

**Syntax:**

**package <identifier>[.<nested ident>[...]]**

**Example:**

```
package telephone;

package telephone.state;
```

# Packages

Java defines a package keyword which will outline to the class which part of the package it resides in. It will self verify on compilation if it exists within the package.

**Syntax:**

**package <identifier>[.<nested ident>[...]]**

**Example:**

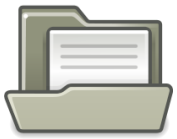
```
package telephone;

package telephone.state;
```

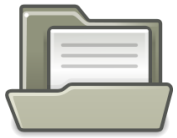
Typically set at the top of your java file, specifies directory it is in.

# Packages

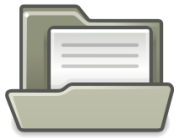
Let's look at the layout of a package



./src

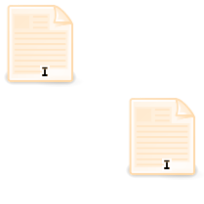


./src/telephone



./src/telephone/state

./src/telephone/input



./src/telephone/state/LineBusy.java

./src/telephone/state/LineWaiting.java

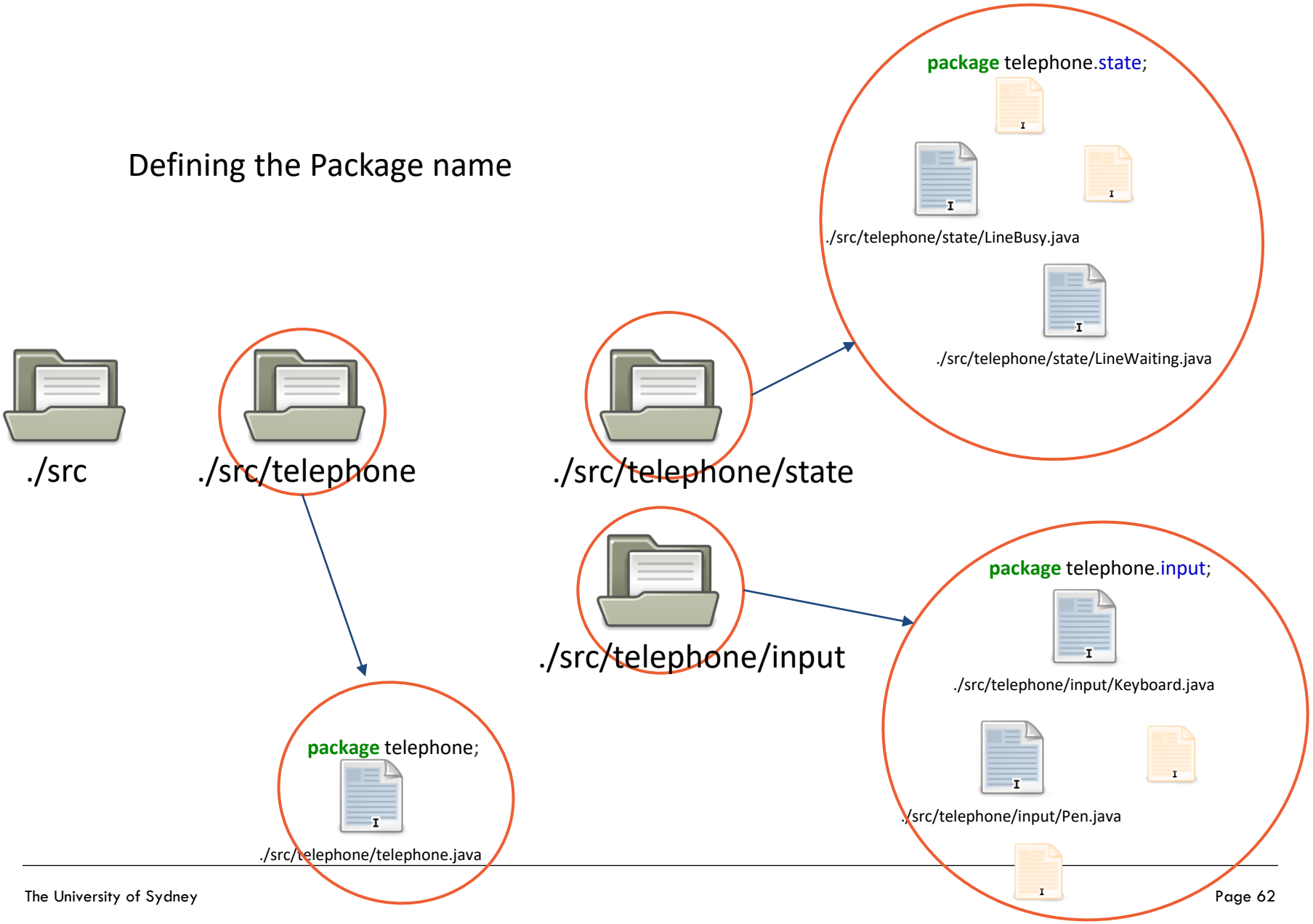
./src/telephone/input/Keyboard.java

./src/telephone/telephone.java

./src/telephone/input/Pen.java

# Packages

Defining the Package name



# Packages

```
package telephone;
public class Telephone {

   private TelephoneState state;

   public Telephone() {
      state = new LineWaiting();
   }

   public void dial(String phonenumber) {
      state = state.dial(phonenumber);
   }

   public void hangup() {
      state = state.hangup();
   }

   public static void main(String[] args) {
      Telephone phone = new Telephone();
      phone.dial("12341234");
      phone.hangup();
   }

}
```

We specify above our above classes and typically above majority of our code, the package name for the file.

# Packages

```
package telephone.state;
public abstract class TelephoneState {

   protected String numberDialed;

   public abstract TelephoneState dial(String phonenumber);

   public abstract TelephoneState hangup();

}
```

```
package telephone.state;
public class LineBusy extends TelephoneState {

   public LineBusy(String number) {
      super();
      numberDialed = number;
   }

   public TelephoneState dial(String phonenumber) {
      throw new InvalidPhoneState();
   }

   public TelephoneState hangup() {
      System.out.println("Hanging up: " + numberDialed);
      return new LineWaiting();
   }

}
```

```
package telephone.state;
public class LineWaiting extends TelephoneState {

   public TelephoneState dial(String phonenumber) {
      System.out.println("Dialing: " + phonenumber);
      return new LineBusy(phonenumber);
   }

   public TelephoneState hangup() {
      throw new InvalidPhoneState();
   }

}
```

We specify the package name within each state class.

# Packages

However! We now need to import these classes into our code so we are able to use them.

```
package telephone.state;
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

We specify the package name within each state class.

```
package telephone.state;
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

```
package telephone.state;
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

# Packages

```
package telephone;
import telephone.state.TelephoneState;
import telephone.state.LineWaiting;

public class Telephone {

   private TelephoneState state;

   public Telephone() {
      state = new LineWaiting();
   }

   public void dial(String phonenumber) {
      state = state.dial(phonenumber);
   }

   public void hangup() {
      state = state.hangup();
   }

   public static void main(String[] args) {
      Telephone phone = new Telephone();
      phone.dial("12341234");
      phone.hangup();
   }
}
```

Our state classes exist is a different package space name, therefore it is unaware they exist.

We will need to import them into our application to utilise them in our code.

# Package Demo

**How could we create an archive?**

# Java Archives

Java provides an archiving format that allows you to compress the files you want to export and distribute to other.

This kind of format is similar to other OS/Package manager specific formats such as **.dmg, .apk, .xdg** and **.deb**.

# Java Archives

To create an archive file, you will need to utilise the **jar** command. We are able to store any kind of data within a java archive but its typical case is bundling and packaging of libraries and applications.

```
> jar -cf MyProgram.jar <list of files>
```

# Java Archives

To create an archive file, you will need to utilise the **jar** command. We are able to store any kind of data within a java archive but its typical case is bundling and packaging of libraries and applications.

```
> jar -cf MyProgram.jar <list of files>
```

Specifies the create and file flag for the **jar** program.

We specify the Jar file to produce and input .class files to be included in the archive.

# Java Archives

.jar Manifest files provide a simple description of requirements your archive files needs.

A common setting is providing an Application Entry point for your .jar file.

By default, creating an archive file will only index the files you have added to it. It will not know what **.class** file you want to execute. You will need to specify that by hand.

# Let’s generate a .jar file

# Build Tools

**Is there a better way?**

Yes! you can look into using the following:

- Apache Ant
- Apache Maven
- Gradle

**Gradle** can be used for building more complex java applications that will involve testing

Each build system intends to make it easier to incorporate libraries, run tests and create multiple application builds.

**See you next time!**

THE UNIVERSITY OF SYDNEY