INFO1113 Object-Oriented Programming

Week 6B: Default Methods and Polymorphism

Copyright Warning

Topics

Default Method

We saw interfaces last lectures and now we will be visiting default methods with java and their utility. This is a new feature in **Java** that allows methods to be defined in an interface.

Prior to Java 8, interfaces just specified the method declaration and never a default method.

Default Method

Default methods are a feature of java 8 and can be difficult to see their use case. In particular, interfaces do not have access to any instance variables and therefore, it may be difficult to see how a default method can be used for reducing complexity of classes.

We are able to depend on the implementation of interface methods that have been declared in the interface. We are then able to utilise the definition given by the concrete type.

Syntax of a default method

Simply we are able to define an **interface** by using the **interface keyword**. Afterwards we will need to define a few

Syntax:

```
 [modifier] default <returntype> MethodName([parameters])
```

Example:

```
            private default void swim();
```

Syntax of a default method

Simply we are able to define an **interface** by using the **interface keyword**. Afterwards we will need to define a few

Syntax:

```
[modifier] default <returntype> MethodName([parameters])
```

Example:

```
private default void swim();
```

Note: the **default** keyword presence is only noted in default methods and **switch** statements.

Default Method

```
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

Default Method

```
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

Liquid container behaviour allows us to fill the container or pour the liquid out of it.

```
public class CoffeeShot implements Liqu
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```
CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

Default Method

```java
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```java
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```java
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

Both **CoffeeShot** and **CoffeeCup** implement this behaviour through the interface.

So where’s the default method?

Default Method

```
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

We can observe that **pour** and **fill** interact with **instance** variables and this makes them a very poor candidate in being generalised to a **default method**.

Default Method

```
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

However, **pourInto** containers certain properties that allow it be ideal in this event. **Why?**

Default Method

```java
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```java
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```java
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

However, **pourInto** containers certain properties that allow it be ideal in this event. **Why?**
It doesn’t interact with instance methods directly and uses interface methods.

Default Method

```
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

We can also see the logic is duplicated between the two classes as well. Considering we want to eliminate duplication, the way we can remove this is through a **default method**.

Let’s demo this

Default Method

```
interface LiquidContainer {
    public void pour(double litres);

    public default pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }

    }

    public void fill(double litres);
}
```

We can move it to the interface and it the method is implemented and can be used by each class that implements **LiquidContainer.**

```
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }

    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }

    public void fill(double litres) {
        this.litres += litres;
    }
}
```

Let’s examine some old code and see if we can reduce the complexity.

Overriding

Overriding allows us to have specify a subtype implementation of the a method.

Overriding applies for **Classes, Abstract Classes** and **Interfaces.** We are able to have a specific implementation that will **always** refer to the subtype implementation, or narrow down to the **last** type that implemented it in the hierarchy.

We are able to override inherited methods and replace them with a subtype specific definition.

Polymorphism

Polymorphism allows us to assert the use of methods specified in an **inherited** type through other types. We saw in previous lecture the use of polymorphism through interfaces and abstract classes.

However we have already seen a number of ways that polymorphism has been employed already.

- Method overloading
- Method overriding

We are able to infer what methods we have access through the type information associated with the object.

Polymorphism - Class

```
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

```
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```
public class Lion {

    public Lion(String name) {
        super(name);
    }

    public void makeNoise() {
        System.out.println("Roar!");
    }

}
```

Polymorphism - Class

```java
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

We have already defined the **makeNoise()** method that majority of cats will make.

```java
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```java
public class Lion {

    public Lion(String name) {
        super(name);
    }

    public void makeNoise() {
        System.out.println("Roar!");
    }

}
```

Polymorphism - Class

```
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

```
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```
public class Lion {

    public Lion(String name) {
        super(name);
    }

    public void makeNoise() {
        System.out.println("Roar!");
    }

}
```

We can see that the **DomesticCat** does not attempt to override this as it is appropriate for the class.

Polymorphism - Class

```
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

```
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```
public class Lion {

    public Lion(String name) {
        super(name);
    }

    public void makeNoise() {
        System.out.println("Roar!");
    }

}
```

However, Lions do not “**meow**” and “**roar**” instead and therefore the noise they make should be different.

Polymorphism - Class

```
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

```
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```
public class Lion {

    public Lion(String name) {
        super(name);
    }

    public void makeNoise() {
        System.out.println("Roar!");
    }

}
```

However, we could treat all 3 classes under the **Superclass**.

Polymorphism with regular classes

```
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

```
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```
public class Lion {
```

```
public static void main(String[] args) {
    Cat[] cats = {new Cat("Felix"),
        new DomesticCat("Garfield"),
        new Lion("Simba")
    };

    for(Cat c : cats) {
        c.makeNoise();
    }
}
```

```
> java CatProgram
Meow!
Meow!
Roar!
```

Final classes and methods

We have the power to inherit from classes but potentially this can open our code up for abuse by other inheriting from classes that shouldn’t be.

We will be introducing a way of stopping classes from being inherited and methods from being overridden.

Syntax of a final method

Similar to the **leaf** class we are able to use the final qualifier as part of the method declaration. Similar to **final** with classes, this infers that the method cannot be overridden.

Syntax:

```
[modifier] final <returntype> MethodName([parameters])
```

Example:

```
private final void swim();
```

Note: the **default** keyword presence is only noted in default methods and **switch** statements.

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;

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

    public String getName() {
        return name;
    }

}
```

```java
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

    public String getName() {
        System.out.println("Infinite Loop! Yeah");
        int i = 0;
        while(i < 10) {}
        return null;
    }
}
```

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;

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

    public String getName() {
        return name;
    }

}
```

```java
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

    public String getName() {
        System.out.println("Infinite Loop! Yeah");
        int i = 0;
                                     {}
```

We have defined two classes, **Person** and **FakePerson** where the latter **overrides** getName().

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;

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

    public String getName() {
        return name;
    }

}
```

```java
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

    public String getName() {
        System.out.println("Infinite Loop! Yeah");
        int i = 0;
        while(i < 10) {}
        return null;
```

We can see that **FakePerson** has overridden **getName()** method with its own implementation.

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;

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

    public String getName() {
        return name;
    }

}
```

```java
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

    public String getName() {
        System.out.println("Infinite Loop! Yeah");
        int i = 0;
        while(i < 10) {}
        return null;
    }
}
```

However we can see the **implementation** is malicious and can allow for execution of arbitrary code.

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user’s code.

Let’s look at the following example:

```
public class Person {
    private String name;
```

```
public class FakePerson extends Person {
```

```
public class PersonProgram {
    public static void sayName(Person p) {
        String name = p.getName();
        System.out.println(name);
    }

    public static void main(String[] args) {
        Person p = new Person("Jimmy");
        FakePerson f = new FakePerson("Mr Evil");
        sayName(p);
        sayName(f);
    }
}
```

```
> java PersonProgram
Jimmy
Infinite Loop! Yeah
```

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

Uh oh! Since it has invoked the sub-types method it will call the infinite loop that has been written.

```java
public class Person {
    private String name;
```

```java
public class FakePerson extends Person {
```

```java
public class PersonProgram {
    public static void sayName(Person p) {
        String name = p.getName();
        System.out.println(name);
    }

    public static void main(String[] args) {
        Person p = new Person("Jimmy");
        FakePerson f = new FakePerson("Mr Evil");
        sayName(p);
        sayName(f);
    }
}
```

```
> java PersonProgram
Jimmy
Infinite Loop! Yeah
```

Don’t let them override the method!

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;

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

    public final String getName() {
        return name;
    }

}
```

We are able to prevent overriding of the method by the subtype by using **final.**

```java
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

    public String getName() {
        System.out.println("Infinite Loop! Yeah");
        int i = 0;
        while(i < 10) {}
        return null;
```

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;
```

```java
public class FakePerson extends Person {
```

```java
public class PersonProgram {
    public static void sayName(Person p) {
        String name = p.getName();
        System.out.println(name);
    }

    public static void main(String[] args) {
        Person p = new Person("Jimmy");
        FakePerson f = new FakePerson("Mr Evil");
        sayName(p);
        sayName(f);
    }
}
```

```
> java PersonProgram
PersonProgram.java:20: error: getName() in
FakePerson cannot override getName() in
Person
    public String getName() {
                  ^
  overridden method is final
1 error
```

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```java
public class Person {
    private String name;
```

```java
public class FakePerson extends Person {
```

```java
public class PersonProgram {
    public static void sayName(Person p) {
        String name = p.getName();
        System.out.println(name);
    }

    public static void main(String[] args) {
        Person p
        FakePerso
        sayName(p
        sayName(f);
    }
}
```

Attempting to override the **getName()** method is prevented by the **compiler**.

```
> java PersonProgram
PersonProgram.java:20: error: getName() in
FakePerson cannot override getName() in
Person
    public String getName() {
                  ^
  overridden method is final
1 error
```

Final classes and methods

When assembling a library we want to ensure that users of the library cannot break it in way that may induce undefined behaviour. This allows us to also differentiate between an errors within a library and user's code.

Let's look at the following example:

```
public class Person {
    private String name;

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

    public final String getName() {
        return name;
    }

}
```

```
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

}
```

We are able to prevent overriding of the method by the subtype by using **final.**

Syntax of a final class

It is best to not be confused with a **constant** class and consider that all attributes are **read-only** once initialised. The final qualifier specifies that the class is a **leaf** class (An endpoint to the hierarchy chain).

Syntax:

```
[modifier] final class ClassName
```

Example:

```
public final class Person
```

Let’s take a look at leaf classes

Scenarios for leaf classes.

We may not want to allow any inheritance due to how the classes have been designed or it may not make any logical sense to extend from such a class.

Let’s use the following cases:

- Inheritance hierarchy of military ranks, should we allow the highest rank to be extended?
- When modelling a **Monarch**, would we extend from the type of the with the largest amount of authority or the lowest?
- A **User** and **Administrator,** should the **Administrator** inherit from **User** or should **User** inherit from **Administrator**?

Organising your application

Classpath

A classpath defines a set of directories exposed to our program. It will allow us to use libraries constructed by others.

We are able to cleanly separate and structure our code into different directories and refer to different segments as part if it was in the same directory.

Classpath

Given a directory that will allow us to store our class file, we will be able to use them directly within our project.

```
> javac -cp .:<Your directory or jar file here>[:<more>]
```

Classpath

You will need to ensure when running your application that your program has access to any class it depended on during compilation.

```
> java -cp .:<Your directory or jar file here>[:<more>]
```

Otherwise your application will not have access to the class files required.

What problems could we encounter when we start importing classes?

Classpath

Potentially two classes can exist with the same name. If this is the case, we have conflicting classes and we will be unable to compile the program with this ambiguity.

For small programs, this is not typically a problem.

It is rare we only work on small programs.

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 com.whiteboard;

package com.whiteboard.render
```

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>[...]]
```

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

Example:

```
package com.whiteboard;

package com.whiteboard.render
```

Demo, using multiple location of class files

Packages

However, there is a drawback with this. As noted before we do not have the ability to distinguish between locations that may have classes of the same name.

This is because these packages are being considered under the **default** package name.

Packages

Let’s look at the layout of a package

./src/whiteboard/render/Drawable.java

./src/whiteboard/render/PositionData.java

./src ./src/whiteboard ./src/whiteboard/render

./src/whiteboard/input

./src/whiteboard/input/Keyboard.java

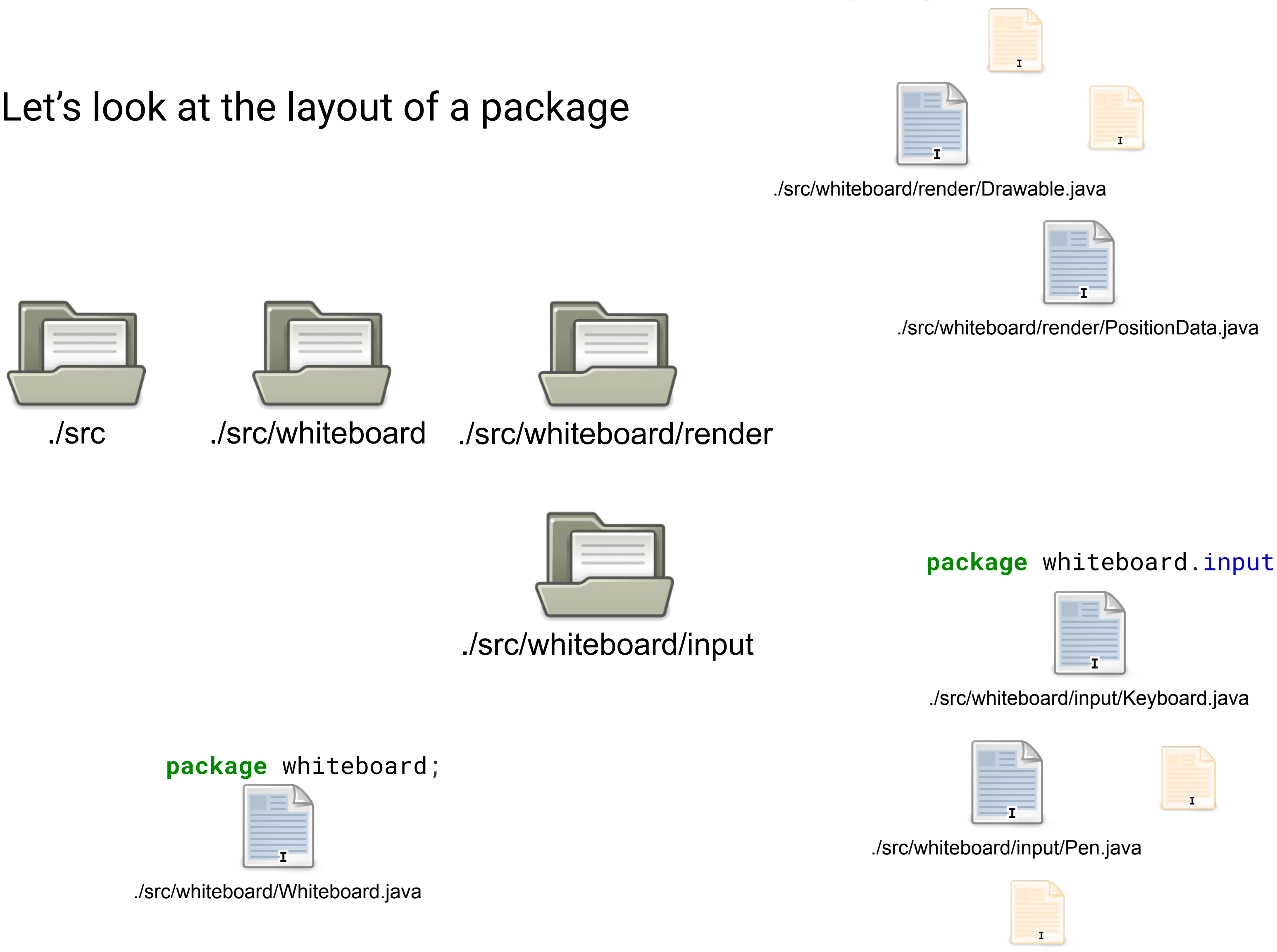
./src/whiteboard/input/Pen.java

./src/whiteboard/Whiteboard.java

Given the current package layout, what would be the package name of each class?

Packages

Let’s look at the layout of a package


./src
./src/whiteboard
./src/whiteboard/render
./src/whiteboard/input
./src/whiteboard/render/Drawable.java
./src/whiteboard/render/PositionData.java
./src/whiteboard/input/Keyboard.java
./src/whiteboard/input/Pen.java
./src/whiteboard/Whiteboard.java

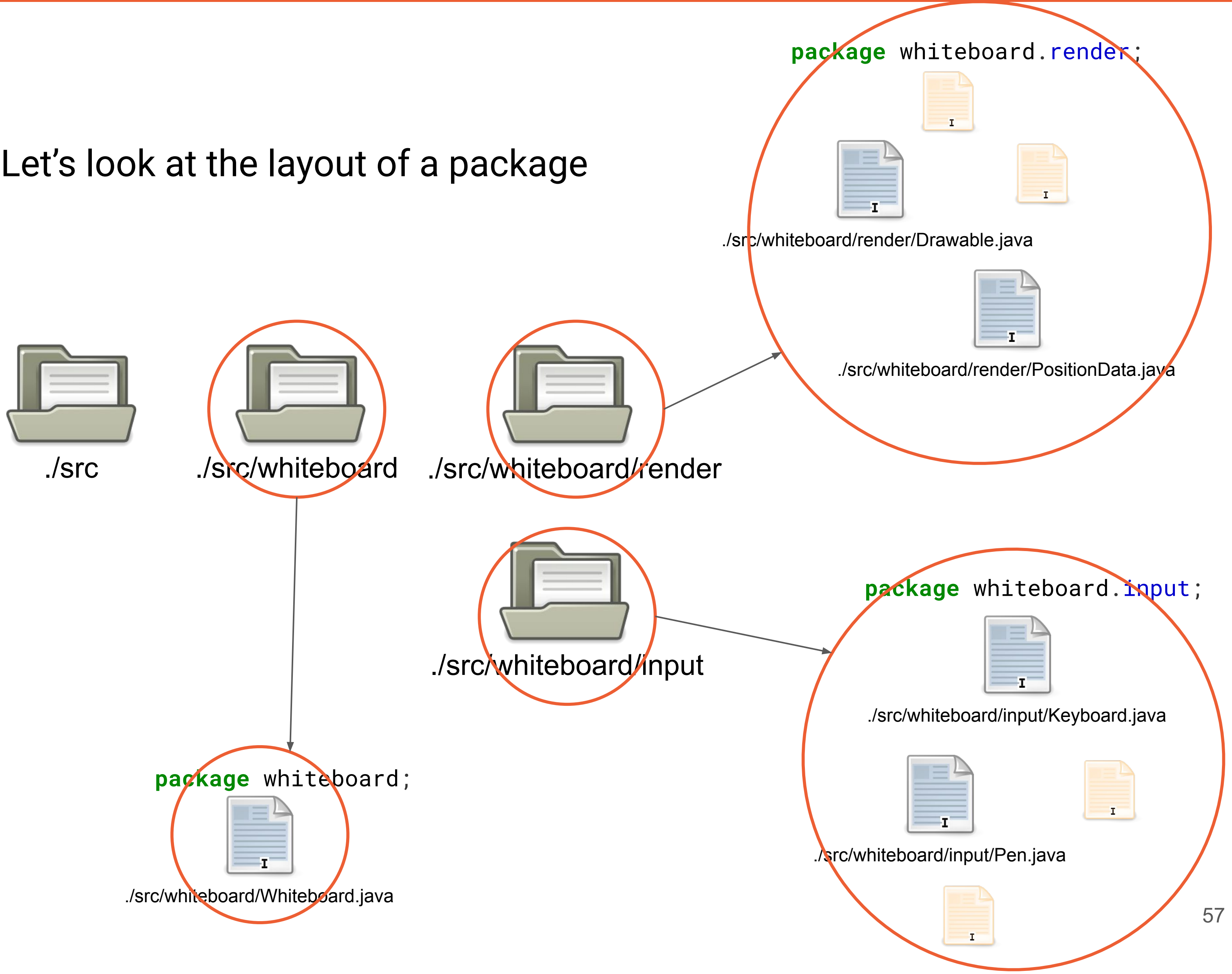
```java
package whiteboard.render;
```

```java
package whiteboard.input;
```

```java
package whiteboard;
```

Packages

Let's look at the layout of a package



./src
./src/whiteboard
./src/whiteboard/render
./src/whiteboard/input
package whiteboard.render;
./src/whiteboard/render/Drawable.java
./src/whiteboard/render/PositionData.java
package whiteboard.input;
./src/whiteboard/input/Keyboard.java
./src/whiteboard/input/Pen.java
package whiteboard;
./src/whiteboard/Whiteboard.java

Packages

So we have laid out the package as the following:

./src

./src/telephone

./src/telephone/state

./src/telephone/exceptions

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. It is **best** practice to put the source file within a folder of the **same name.**

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.

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

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

```java
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();
    }

}
```

```java
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

```
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.

How could we create an archive?

Packages

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**.

Packages

.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.

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.

Let’s generate a .jar file

Build Tools

Yes! Although they are outside the scope of this course, you can look into using the following:

- Apache Ant
- Apache Maven
- Gradle

We will be using **Gradle** 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!