Java Object Oriented Approach

Interfaces

Java Object-Oriented Approach

Java Object-Oriented Approach

- Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)
- Define and use fields and methods, including instance, static and overloaded methods
- Initialize objects and their members using instance and static initialiser statements and constructors
- Understand variable scopes, apply encapsulation and make objects immutable
- Create and use subclasses and superclasses, including abstract classes
- Utilize polymorphism and casting to call methods, differentiate object type versus reference type
- Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods
- Create and use enumerations

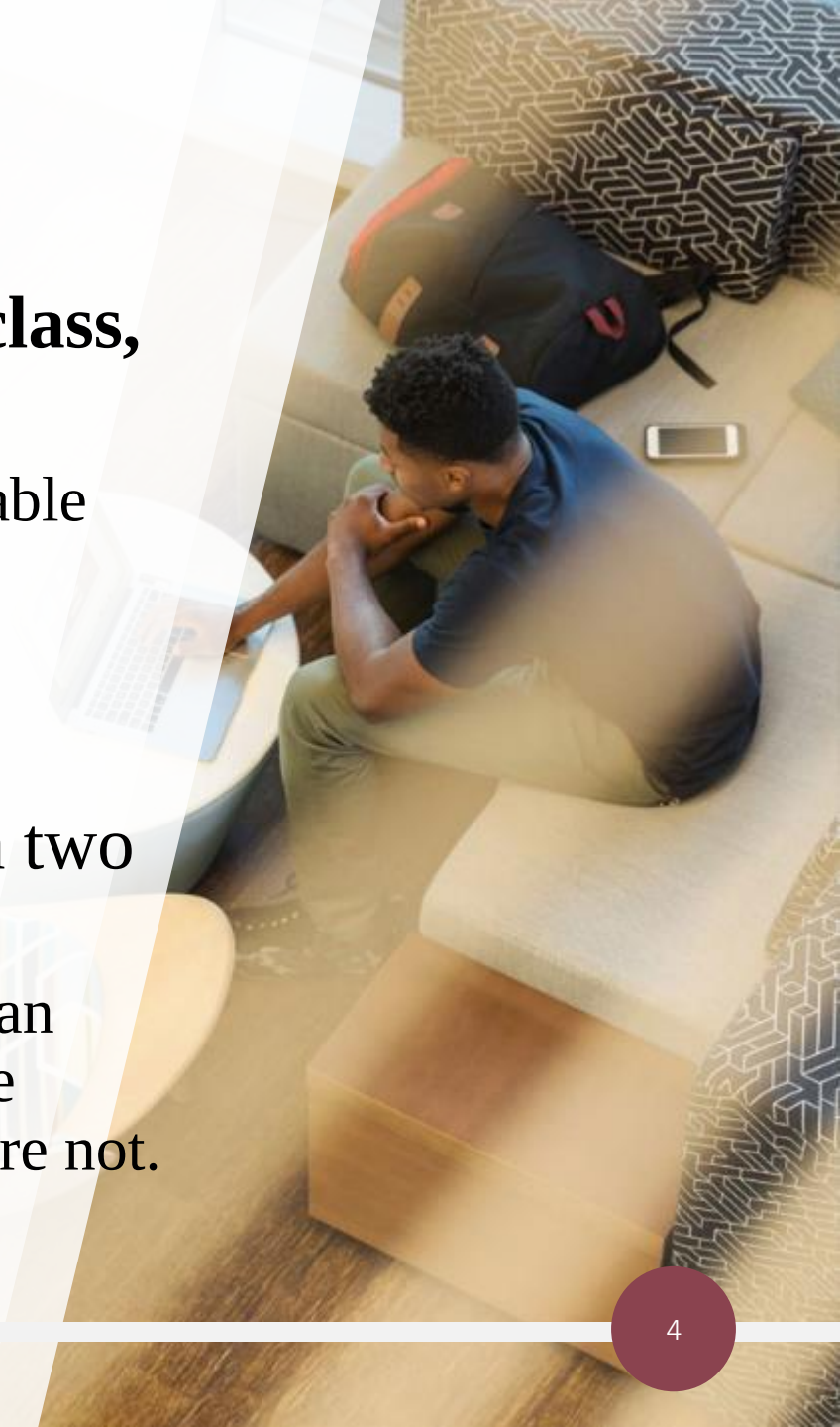
Interfaces

- In general, when you create an interface, you are defining a contract for *what* a class can do; without saying anything about *how* the class will do it.
- A class “signs” the contract with the keyword *implements*.
- When implementing an interface, you are agreeing to adhere (obey) to the contract defined in the interface.
- If a concrete (non-abstract) class is implementing an interface, the compiler will ensure that the class has implementation code for each abstract method in the interface.

Interfaces

- **Whereas a class can extend from only one other class, a class can implement many interfaces**.
  - class Dog extends Animal implements Moveable, Loveable
  - a Dog “is-a” : Animal, Moveable and Loveable

- As of Java 8, it is now possible to inherit **concrete** methods from interfaces. Interfaces can now contain two types of concrete methods: *static* and *default*.
  - Implementation classes are NOT required to implement an interface’s *static* or *default* methods. The *default* interface methods are inheritable but the *static* interface methods are not.



Interfaces

- Interfaces themselves are implicitly *abstract*
  - public abstract interface I{} ⇔ public interface I
  - top-level interfaces can have *public* or package-private access

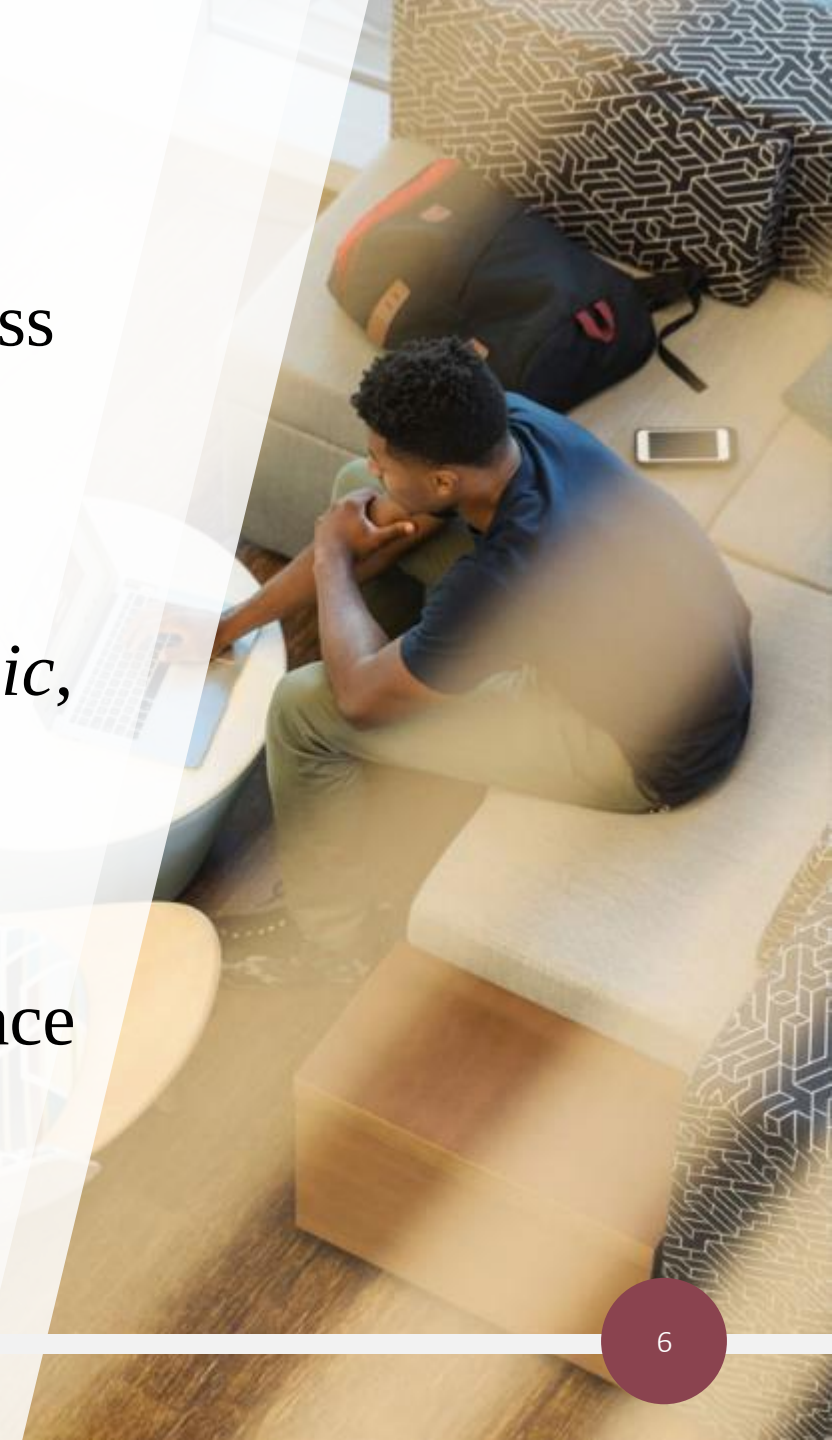
- All interface methods are implicitly *public*

Interfaces

- All interface methods are implicitly *abstract* (unless declared as *default* or *static*, the new features introduced in Java 8).

- All variables declared in an interface must be *public*, *static* and *final* i.e. interfaces can only declare constants (not instance variables).

- As with *abstract* classes you cannot *new* an interface type but they can be used as references:
  - *Printable p = new Printer(); // Printable is an interface*



Interface Methods (abstract examples)

Ok

```
public interface I {
    // all of the following methods
    // are legal and identical
//    void m();
//    public void m();
//    abstract void m();
//    public abstract void m();
     abstract public void m();
}
```

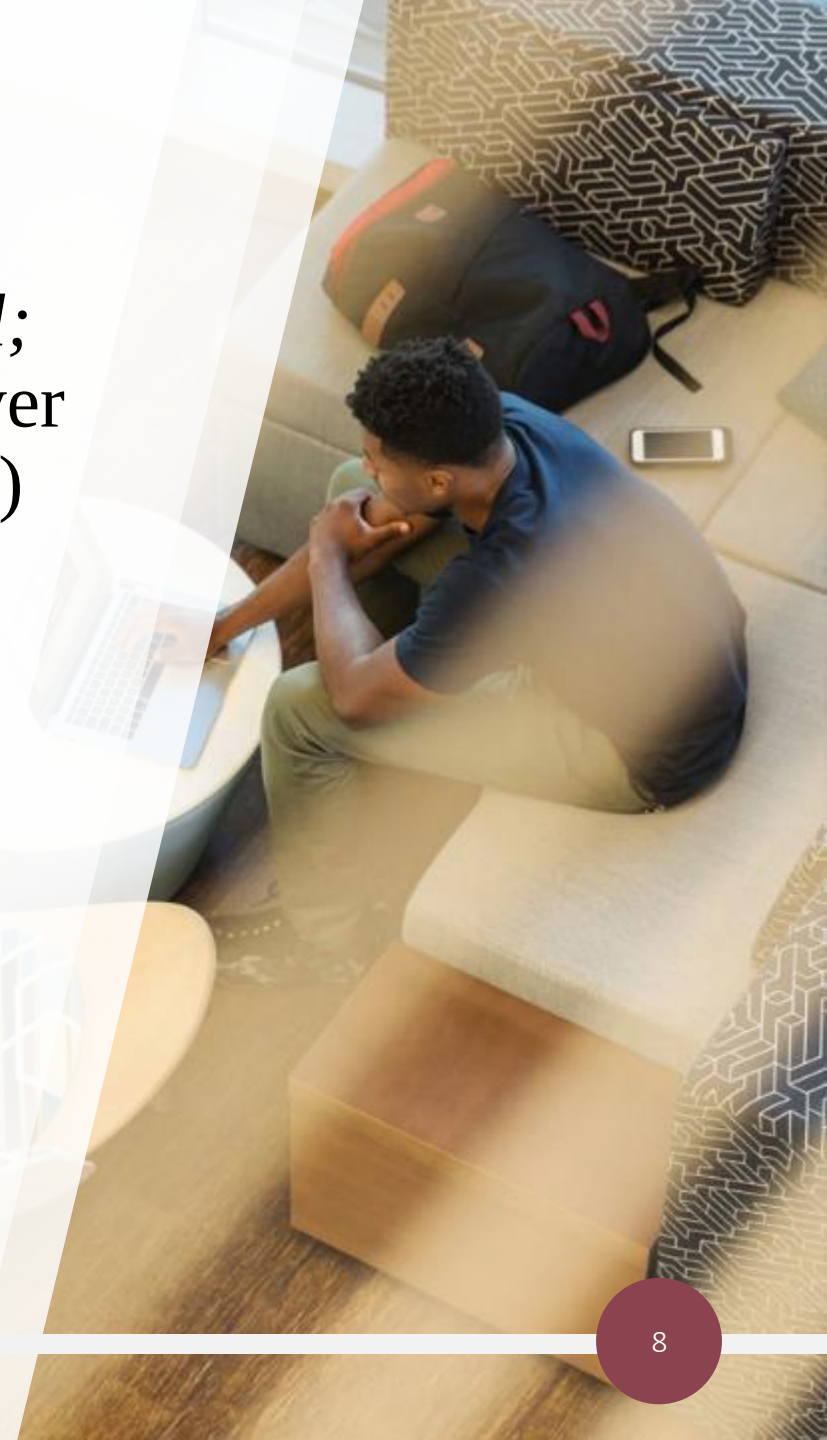
Errors

```
interface I2{
//    final void m();// final and abstract not allowed
//    private void m();// must be public
    protected void m();// must be public
}
```

Interface Constants

- The key rule: by default they are *public static final;* this means that they are read-only i.e. they can never be given a value by an implementing (or any other) class.

- By placing the constants in the interface, any class implementing the interface has direct access to the constants, just as if the class had inherited them.

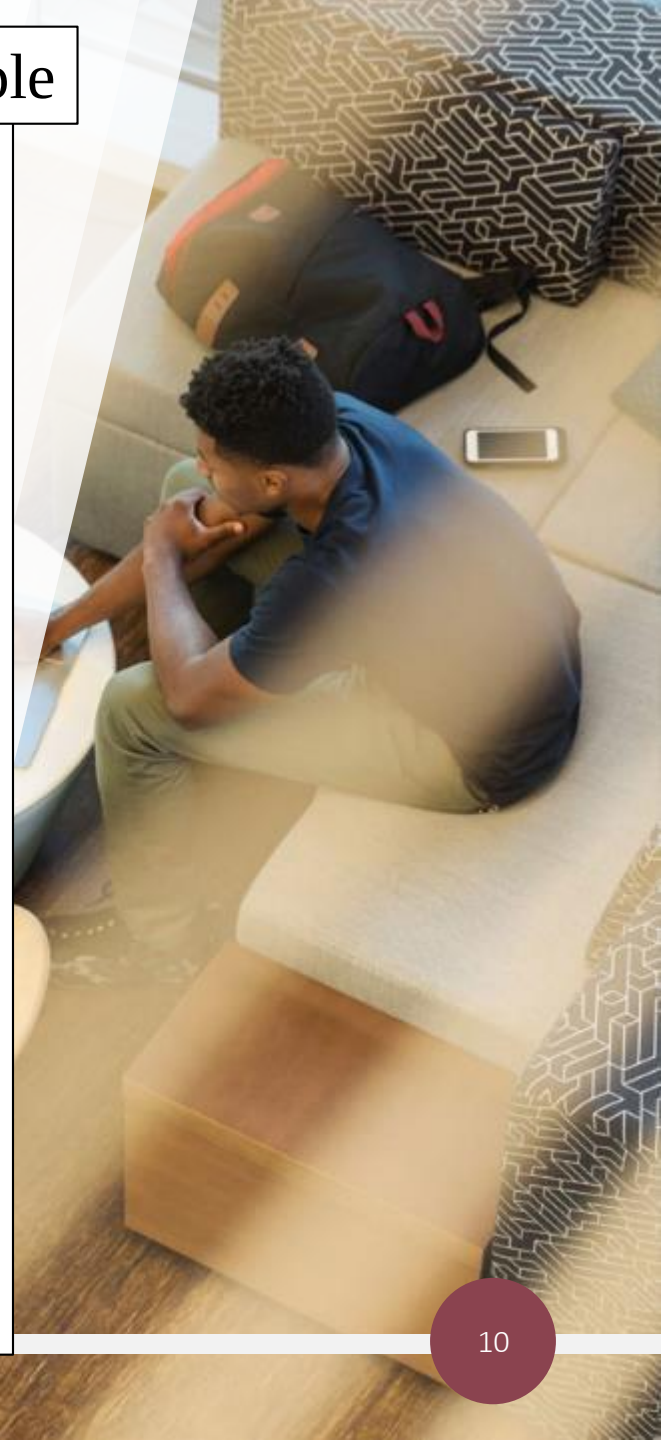


Interface Constants

```
interface I3{
//     int x=1;                          // public static final
//     public int x=1;                   // static final
//     static int x=1;                   // public final
//     final int x=1;                    // public static
//     public static int x=1;            // final
//     public final int x=1;             // static
//     static final int x=1;             // public
//     public static final int x=1;        // what you get implicitly
     final static public int x=1;        // order not NB
}
```

Example

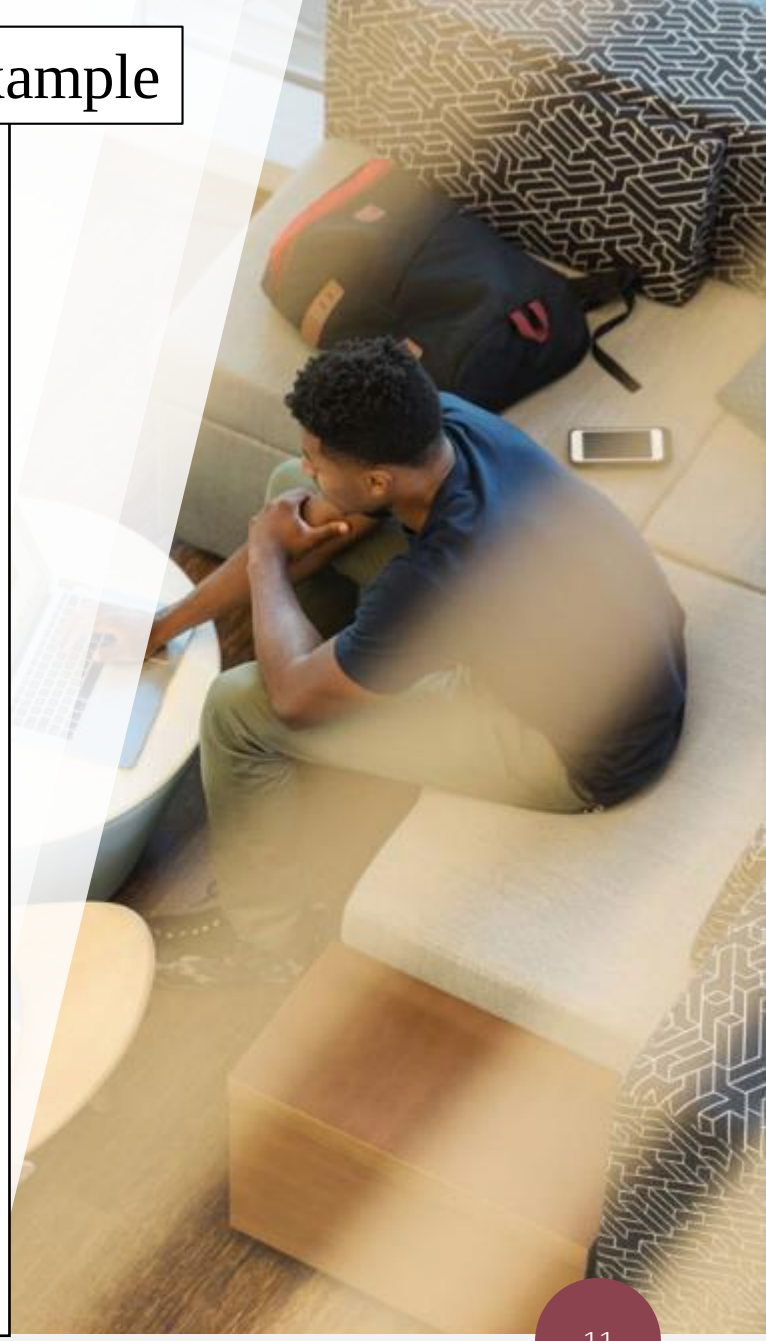
```
interface Moveable{
    String s="move";// constant - public static final
    void move(); // public abstract
}
public class Dog implements Moveable{
    // MUST be public - cannot assign weaker privileges
//     void move(){}
    @Override
    public void move(){// MUST be public
        System.out.println("Dog::move()");
    }
    public static void main(String[] args) {
//        s = "walk"; // cannot change a final variable
        System.out.println(s);// move
        // cannot refer to an instance member from a static context
//        move();
        new Dog().move();// Dog::move()
    }
}
```



Example

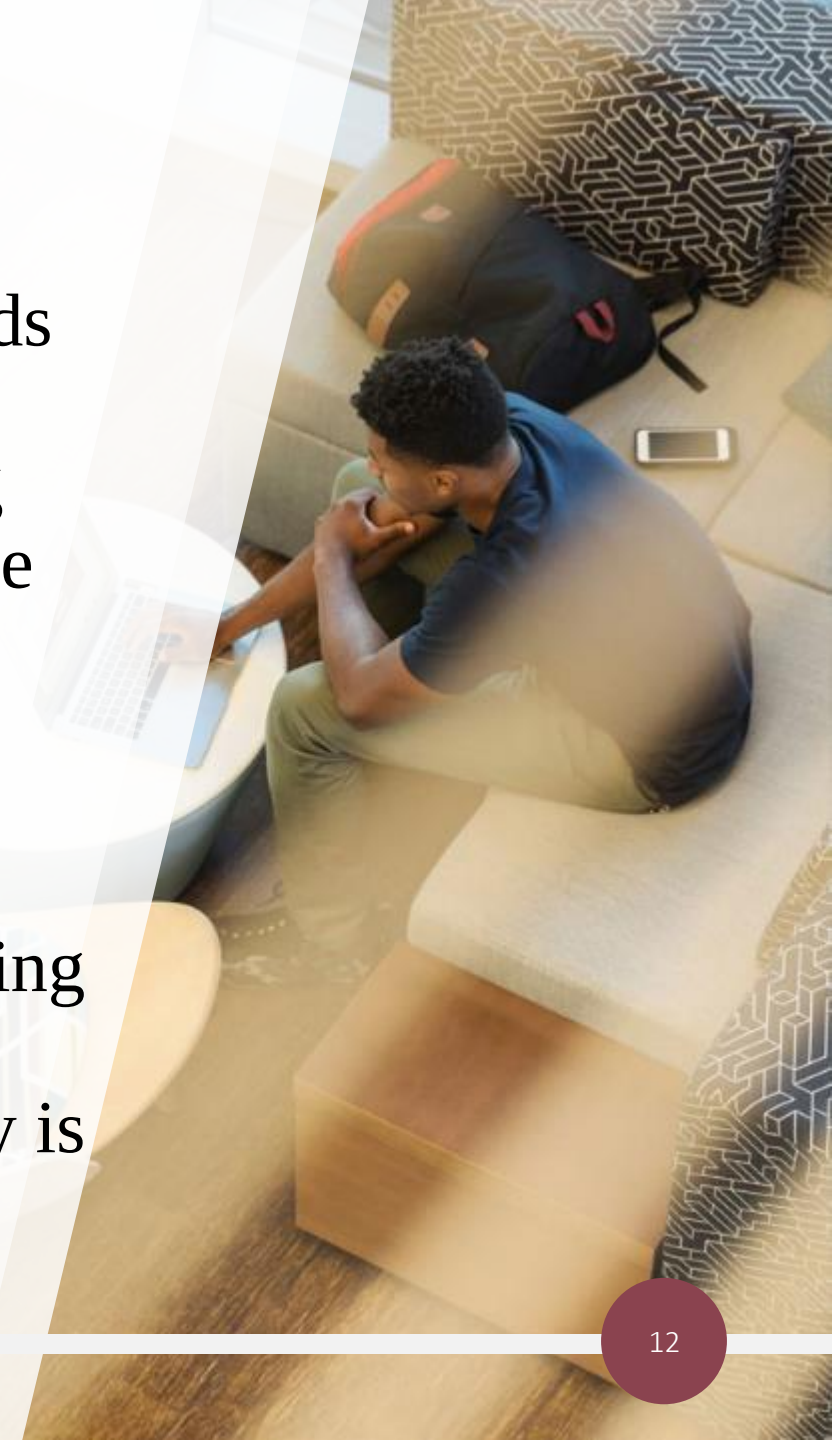
```
interface Moveable{
    void moveIt();
}
interface Spherical{
    void doSphericalThings();
}
// interfaces can extend from more than one interface
interface Bounceable extends Moveable, Spherical{
    void bounce();
}
// Concrete class VolleyBall must implement all abstract
// methods in Bounceable
class VolleyBall implements Bounceable{
    @Override public void moveIt(){}
    @Override public void doSphericalThings(){}
    @Override public void bounce(){}
}
// BeachBall is ok because it is abstract - can implement
// some, all or none of the abstract methods in Bounceable
abstract class BeachBall implements Bounceable{

}
```



default Interface Methods

- As of Java 8, interfaces can include *default* methods with concrete implementations. Why? Previously, adding new methods to an interface broke existing clients as interfaces required all implementing code to provide implementations for all the methods (which would include the new ones). Default methods allow us to deal with this issue - new *default* methods (along with their implementation code) are automatically available in all implementing code. Thus, there is no need to refactor the client implementing classes and backwards compatibility is maintained.



default Interface Methods

- *default* methods use the *default* keyword and *default* methods are not allowed in classes, only in interfaces.
- *default* methods are *public* by default (*public* is optional).
- *default* methods **must** have a concrete method body i.e. {}.
- Implementation classes are NOT required to implement an interfaces *default* (or *static*) methods.

default examples

```
package lets_get_certified.java_ooa;

public interface TestDefault {
    int m0(); // ok, regular abstract method
    default int m1() {return 4;} // ok
    public default void m2() {}  // ok, public by default

    // default and static not allowed together
    default static void m3(){}

    // missing method body i.e. {}
    public default void m6();

    // default expects a method body, abstract does not
    default abstract void m7();
}
```

Example

```
interface Moveable{
    default void move (){
        System.out.println("Moving");
    }
}
class Cheetah implements Moveable{
    @Override
    public void move (){
        System.out.println("Moving very fast!");
    }
}
class Elephant implements Moveable{}
public class TestAnimal {
    public static void main(String[] args) {
        // cannot new an interface type
        //Moveable m1 = new Moveable();
        Moveable cheetah = new Cheetah();
        cheetah.move();// Moving very fast!
        Moveable elephant = new Elephant();
        elephant.move();// Moving
    }
}
```

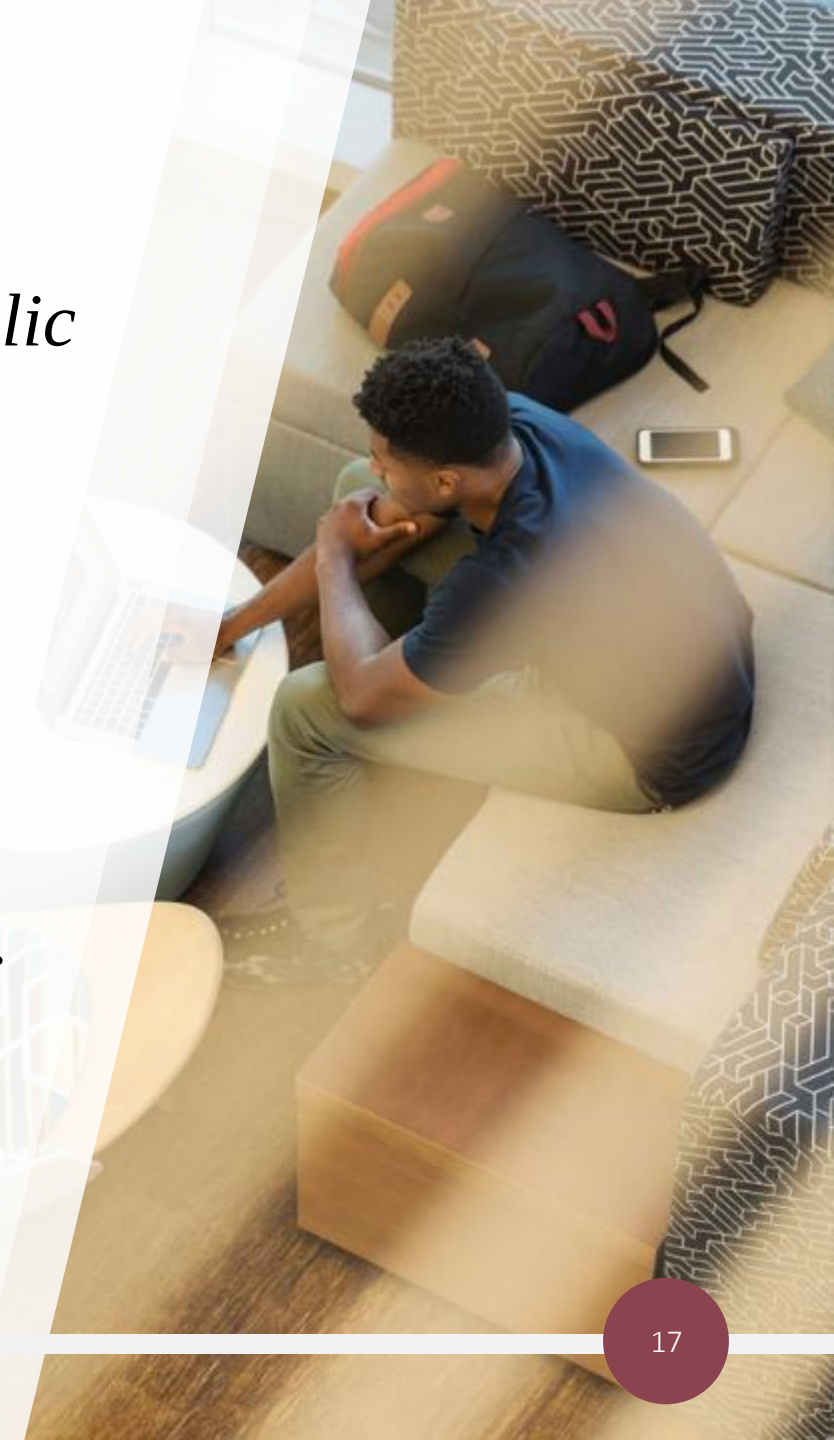
static Interface Methods

- As of Java 8, interfaces can include *static* methods with concrete implementations. Now, interface-related utilities and factory methods can be stored *in* the interface instead of needing separate classes thus "*making it easier to organise your helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class*" [Java Docs].

- *static* interface methods use the *static* keyword.

static Interface Methods

- *static* interface methods are public by default (*public* is optional).
- *static* interface methods must have a concrete method body i.e. {}.
- When invoking a *static* interface method, the interface name must be included in the invocation.



static examples

```
interface StaticIface{
    static int m1() { return 4;}
    public static void m2() {}
//    static void m5(); // missing method body
}

class TestSIF implements StaticIface{
    public static void main(String []args){
        System.out.println(StaticIface.m1()); // 4
//        System.out.println(m1()); // error - need "interface_name.m1()"

        new TestSIF().go();
    }
    public void go(){ // works from an instance method too!
        System.out.println(StaticIface.m1()); // 4
    }
}
```

static methods are not inherited

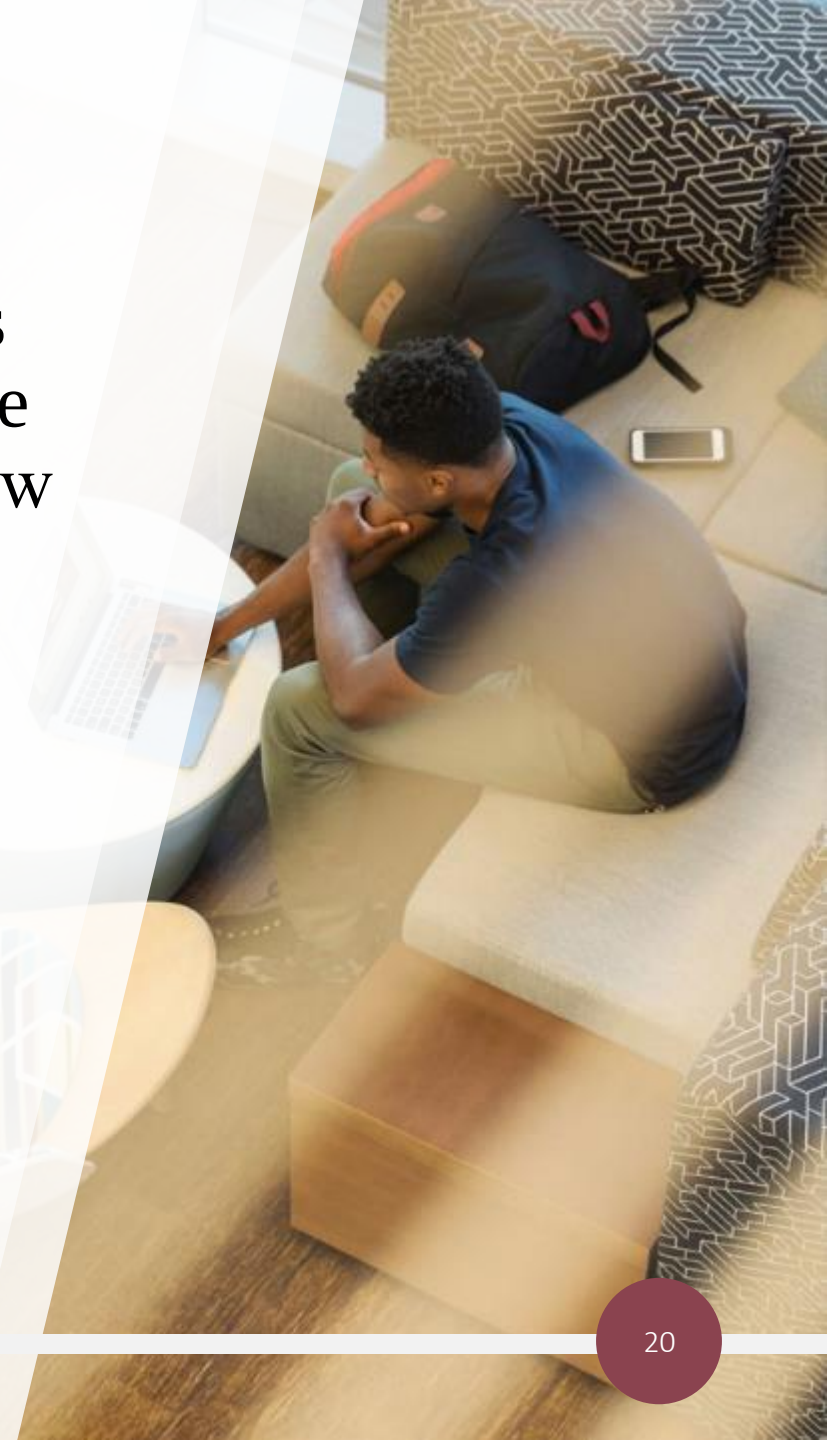
```
interface Moveable{
    static void stand(){}
    default void move(){
        System.out.println("moving...");
    }
}
class Cheetah implements Moveable{
    @Override
    public void move(){// cannot override unless it is inherited!
        System.out.println("moving very fast!");
    }

    @Override
    public void stand(){}
}
```

```
method does not override or implement a method from a supertype
----
(Alt-Enter shows hints)
```

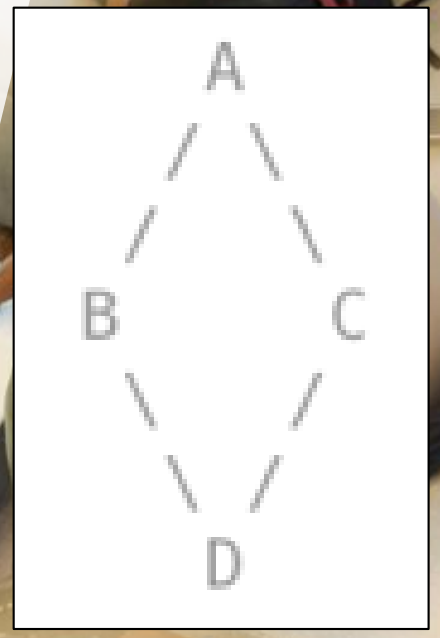
Multiple Inheritance

- In languages such as C++, multiple inheritance is supported i.e. where a class can extend from more than one class. Java's creators decided not to allow this as it can lead to a scenario such as the "Diamond of Death".
- Diamond of Death
  - gets its name from the UML class diagram representing the problem



Multiple Inheritance

- Diamond of Death
  - the diamond is formed when classes B and C extend from A; in addition, D extends from both B and C - A is at the top of the diamond and D is at the bottom of the diamond
  - if class A defines a method *m()* and B and C both override *m()* then D has inherited two different implementations of *m()*



Multiple Inheritance

- Given that:
  a) Java supports multiple interface implementation
  b) Java 8 introduced concrete methods into interfaces

is the Diamond of Death possible?

- No, if you want to code a class that implements interfaces that have identical concrete methods, the compiler ensures that you override the method in the class you are coding.

Multiple Inheritance

```
interface I1{
    default int m(){return 1;}
}
interface I2{
    default int m(){return 2;}
}
public class MultipleInheritanceTest implements I1, I2 {
    public static void main(String []args){
        new MultipleInheritanceTest().go();
    }
    void go(){
        System.out.println(m());
    }
    // it compiles once the following method m() is commented in
//    @Override
//    public int m(){
//        return 3;
//    }
}
```

Multiple Inheritance

- Sample program