Java Object Oriented Approach

Inheritance

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

Inheritance

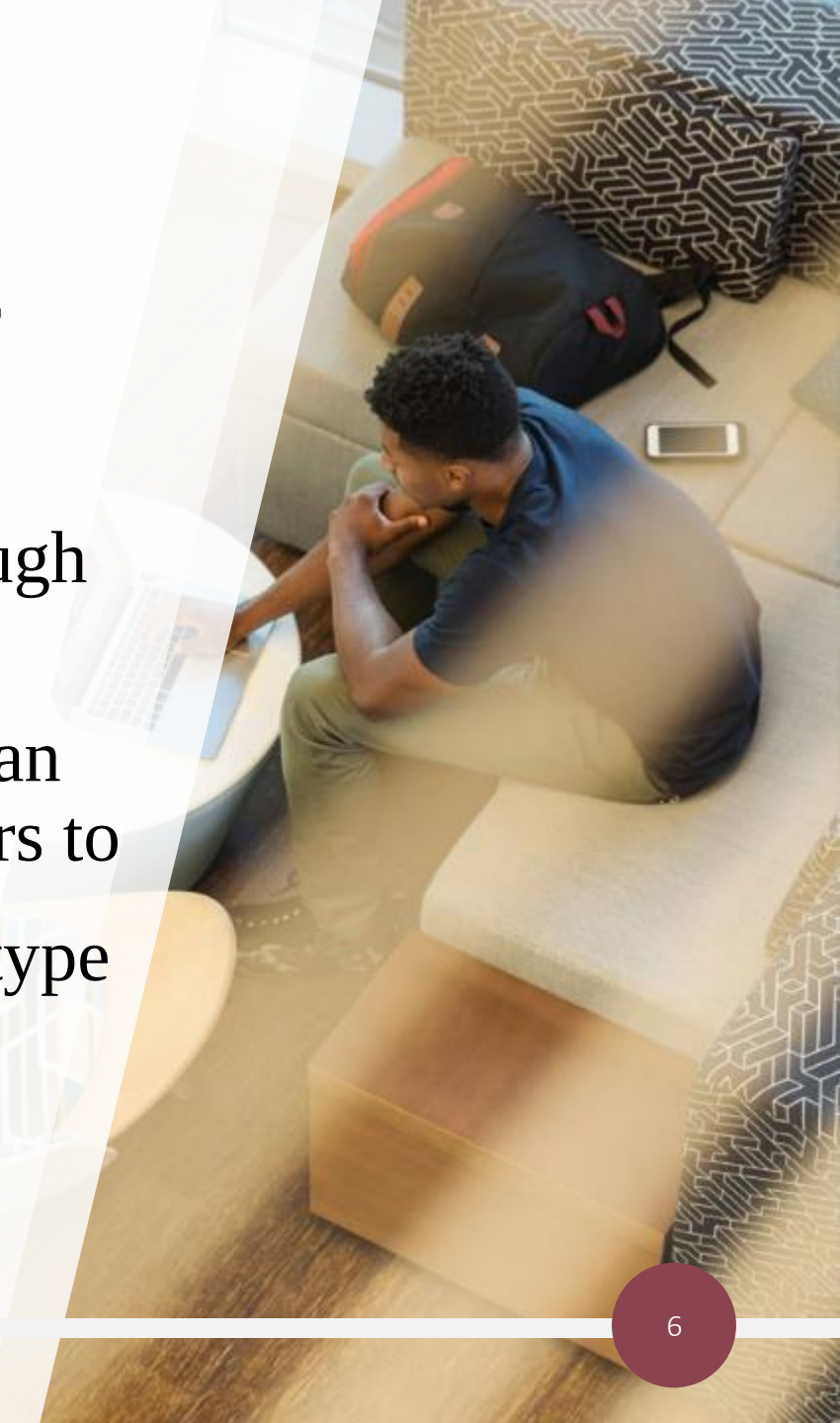
- A core pillar in OOP.
- Inheritance is a code reusability mechanism in Java in which common properties between related types are exploited by forming relationships.
- The common properties are provided in the *super/parent/base* class and are available to the more specialised *sub/child/derived* class.
- Inheritance relationships in Java are created by extending from a class (*extends*) or implementing an interface (*implements*).
- Inheritance forms an “IS-A” relationship.

Polymorphism

- Another core pillar in OOP.
  - "PIE" = Polymorphism, Inheritance and Encapsulation
- The term polymorphism has Greek origins: "poly" (many) and "morphe" (forms).
- Any Java object that can pass more than one IS-A test can be considered polymorphic. Other than objects of type *Object*, all Java objects are therefore polymorphic (as they pass the IS-A test for both *Object* and their own type).

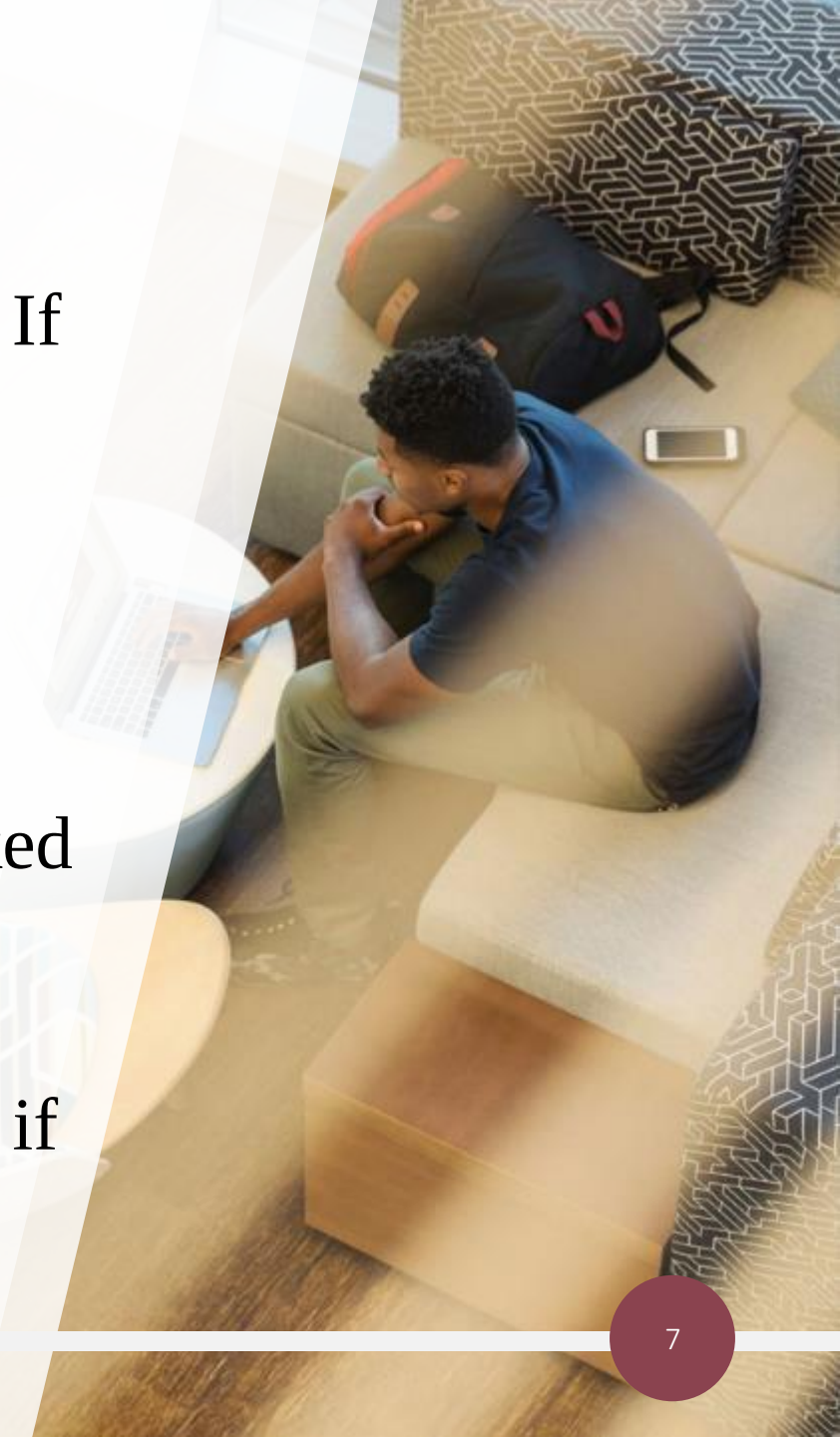
Polymorphism

- *Car c* = new Car(); // c is the reference; its type is Car
- Some key points about references:
  - the reference type can never be changed (although the object it refers to can change)
  - the reference type determines which methods can be invoked on the object that the reference refers to
  - a reference can refer to any object of the same type as the references' type or any subtype of the references' type i.e. across and down in the inheritance tree



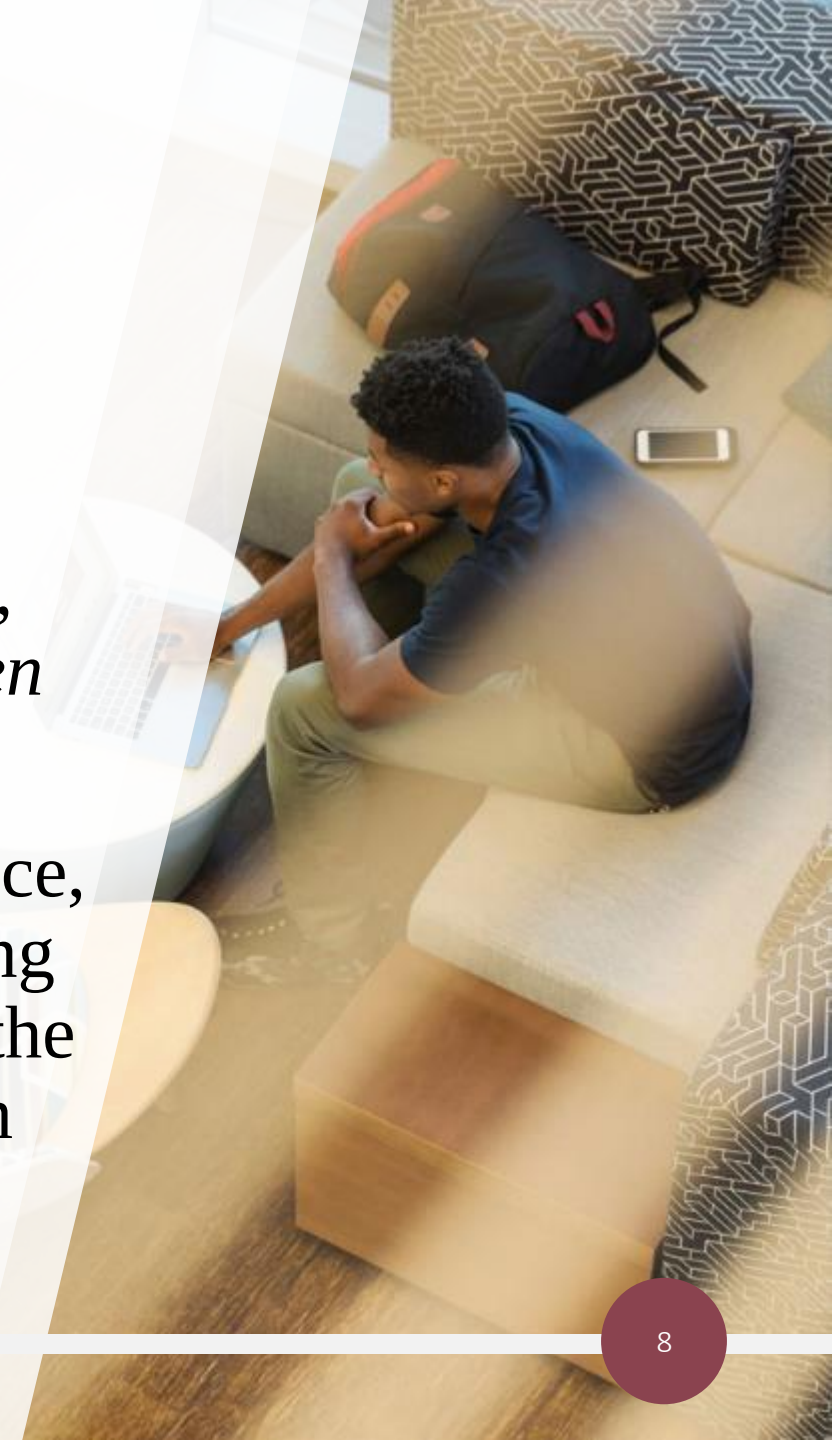
Polymorphism

- A reference can be a class type or an interface type. If the reference is an interface type, the reference can refer to any object of any class that *implements* the interface.
- Polymorphic method invocations apply only to *instance methods* i.e. non-static methods. Only overridden instance methods are dynamically invoked based on the object.
- If at runtime, you are accessing data (instance or *static*) or *static* methods, the reference type is used; if its an instance method, the object type is used.



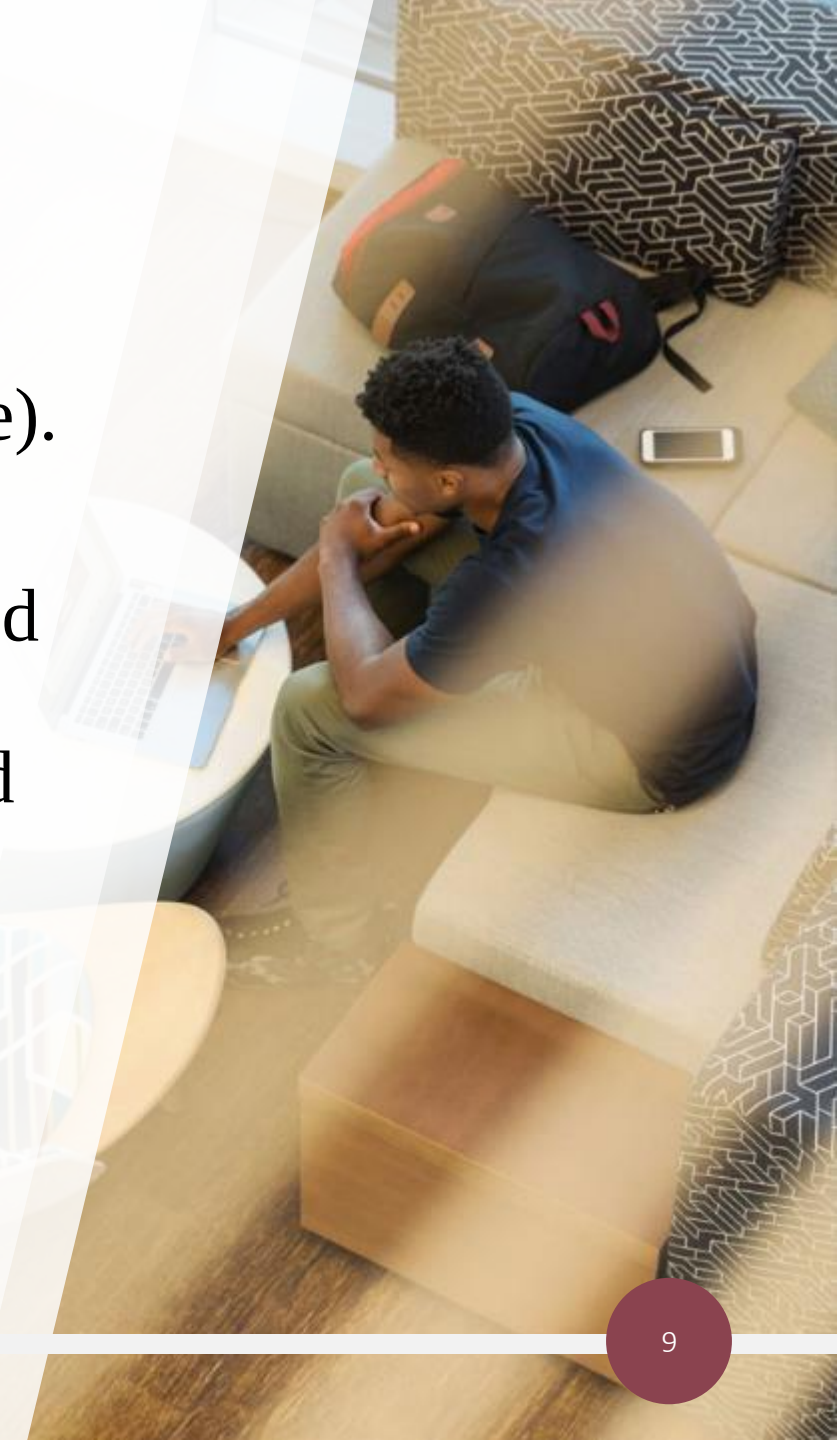
Overriding/Overloading

- At compile time, it is the reference type that is important in determining which *overloaded* methods, depending of their signatures will be invoked.
- At runtime, polymorphism kicks in and the JVM, based on the object type, selects which *overridden* method to execute.
- Overriding - remember that, because of inheritance, a subtype is guaranteed to be able to do everything the super type can do. The subtype can override the instance method and thus provide its own custom implementation; or, the subtype can just use the super type version it inherited.



Overriding/Overloading

- Overloading - where you can reuse the same method name in a class but with different arguments (and optionally, a different return type).
- Which *overridden* version of the method to call (i.e. which class in the inheritance tree) is decided at *runtime* based on the *object* type, but which *overloaded* version of the method to call is based on the *reference* type at *compile* time.
- Sample toString() program.



Overriding/Overloading

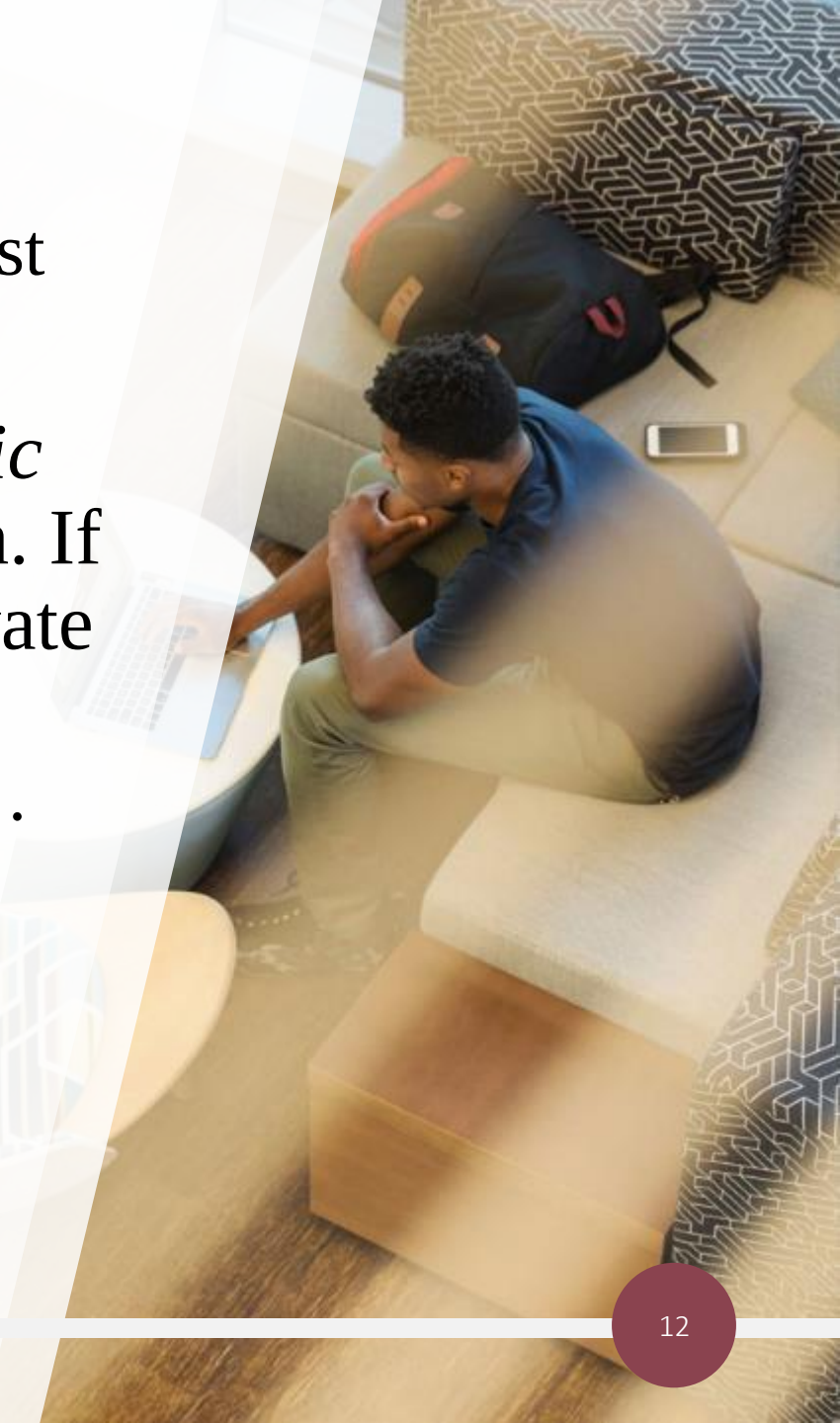
```
class Animal{
    public void eat(){System.out.println("Animal::eat()");}
    public void eat(String s){System.out.println("Animal::eat(String)");}
}
class Horse extends Animal{
    @Override
    public void eat(){System.out.println("Horse::eat()");}
    public void buck(){System.out.println("Horse::buck()");}
}
public class TestAnimals {
    public static void main(String []args){
        Animal aa = new Animal();
        aa.eat();            // Animal::eat()
        aa.eat("something");// Animal::eat(String)
        aa.buck();  // "buck()" not in Animal class

        Animal ah = new Horse();
        ah.eat();            // Horse::eat()
        ah.eat("something");// Animal::eat(String) - inherited method
        ah.buck();  // "buck()" not in Animal class

        new Horse().buck();// Horse::buck()
    }
}
```

Overriding a method rules

- The argument list must exactly match that of the overridden method; otherwise you are just overloading the method.
- The return type must be the same as, or a subtype of (covariant return), the return type declared in the overridden method in the superclass.
- The access level cannot be more restrictive that that of the superclass method.

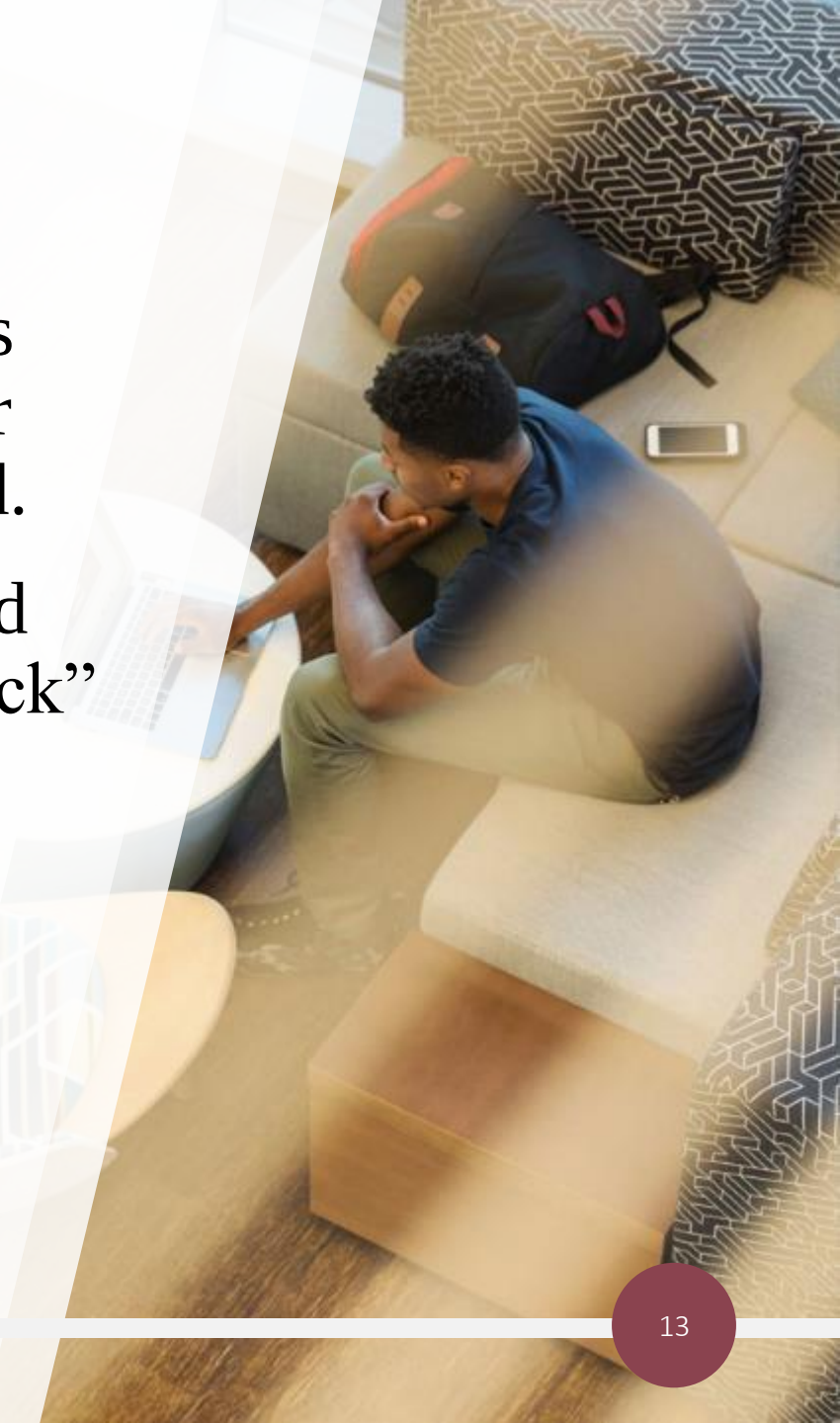
Overriding a method rules

- Remember that the method must be inherited first before it can be overridden.
  - If the parent class method is *private*, *static* or *final*, the method cannot be overridden. If the parent class method has package-private access and you are trying to override the method in a class in a different package… the method cannot be overridden.



Overriding a method rules

- The overriding method cannot throw new or broader checked exceptions than the parent class method. The overriding method can throw fewer checked exceptions than the parent class method.
- The overriding method can throw any unchecked exception it likes (as the compiler does not “check” unchecked exceptions).

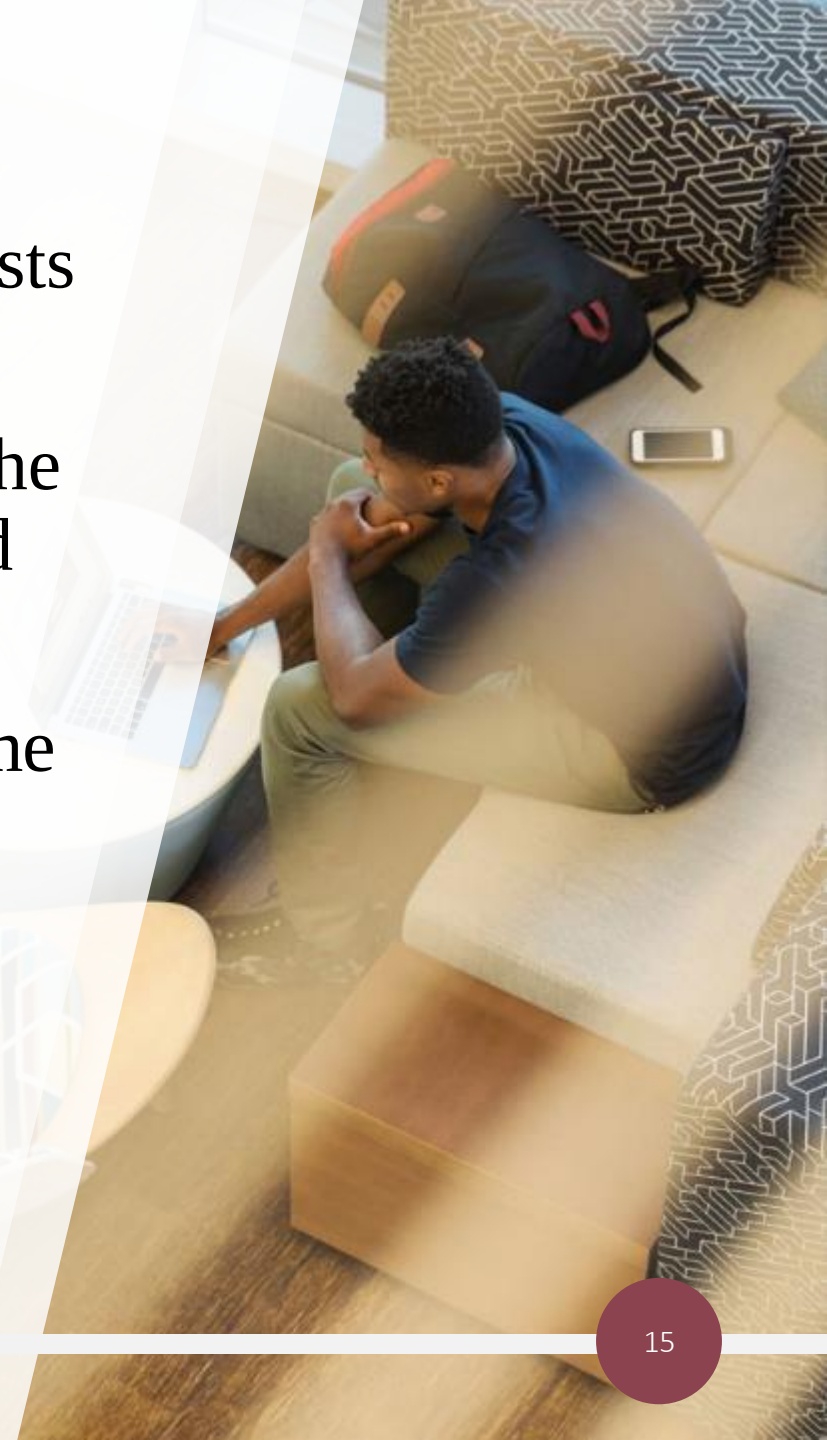


Illegal overrides

```
class Animal {
    public void eat(){}
    public Animal drink(){ return new Animal();}
}
class Cow extends Animal {
    private void eat(){} // access modifier cannot be weaker
    public void eat() throws IOException{}  // cannot throw new
                                            // or broader exceptions
    public void eat(String s){} // an overload, not an override
    public String eat(){} // return type should be void

    @Override   // covariant return types
//    public Animal drink(){ return new Animal();} // ok
    public Cow drink(){ return new Cow();} // ok
}
```

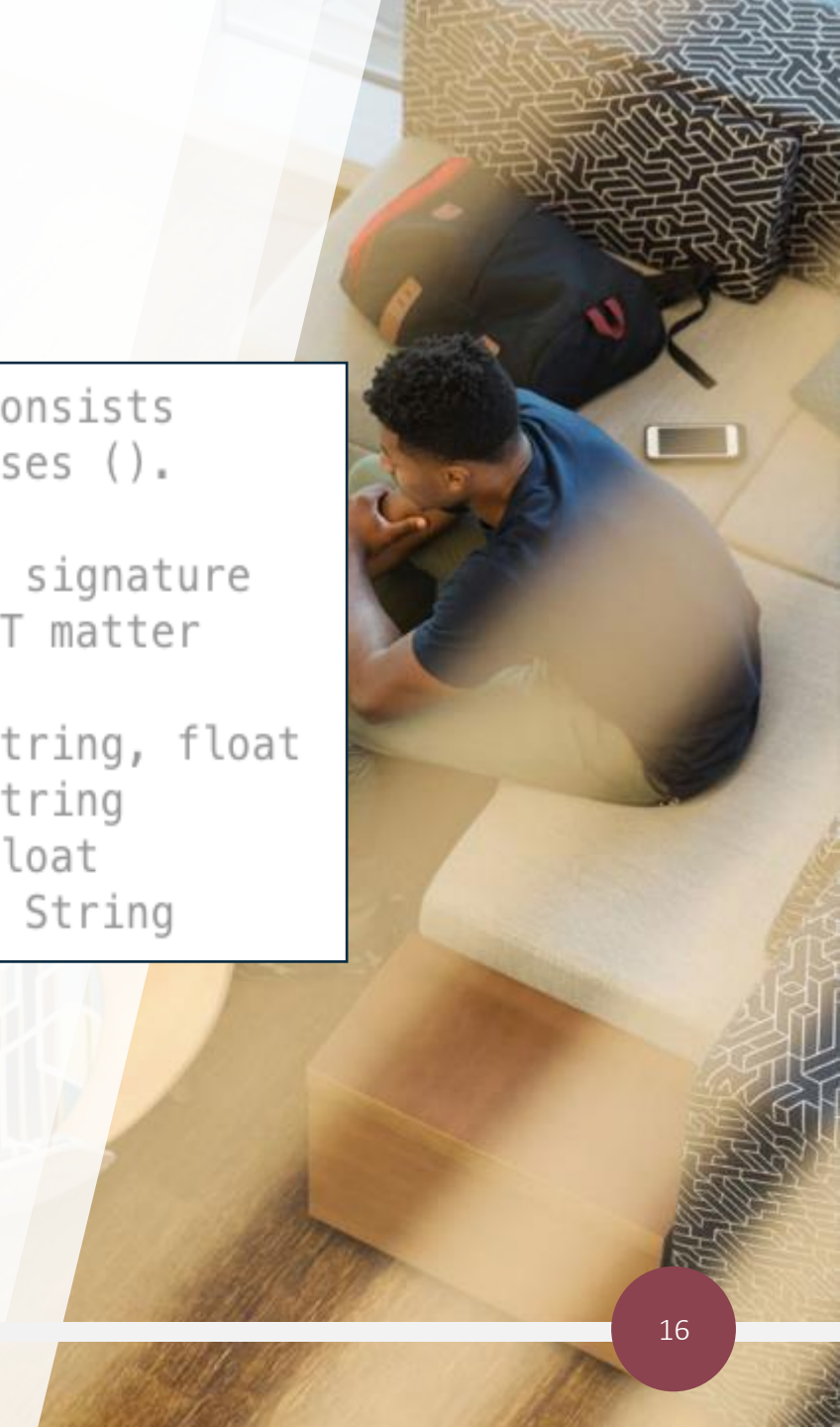
Overloading a method rules

- Overloaded methods MUST change the argument lists (either in argument type or the order of the types).
- Overloaded methods CAN change the return type, the access modifier and declare new or broader checked exceptions.
- An overloaded method can be overloaded in the same type or in a subtype.



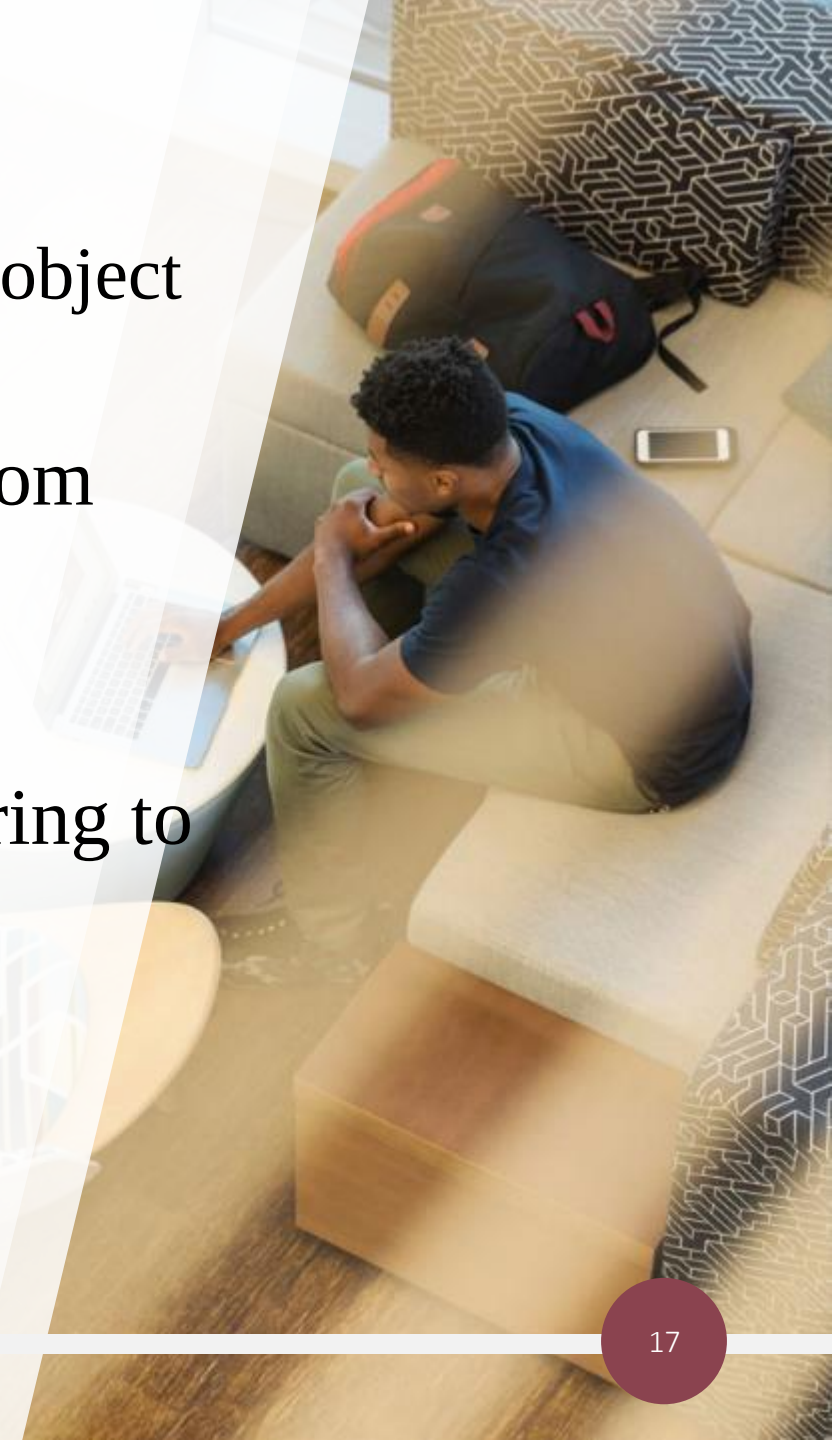
Overloading a method rules

```
// 1. Argument lists MUST be different. The method signature consists
//    of the method name and the arguments inside the parentheses ().
//    The names of the arguments does not matter at all.
// 2. The return type, the access priveleges and the exception signature
//    are NOT part of the method signature and therefore do NOT matter
//    from the viewpoint of method overloading.
public void m(int i, String s, float f){}              // int, String, float
public void m(int i, String s){}                       // int, String
private int m(int i, float f){return 1;}               // int, float
public void m(float f, String s) throws IOException{}// float, String
```



instanceof and *equals(Object)*

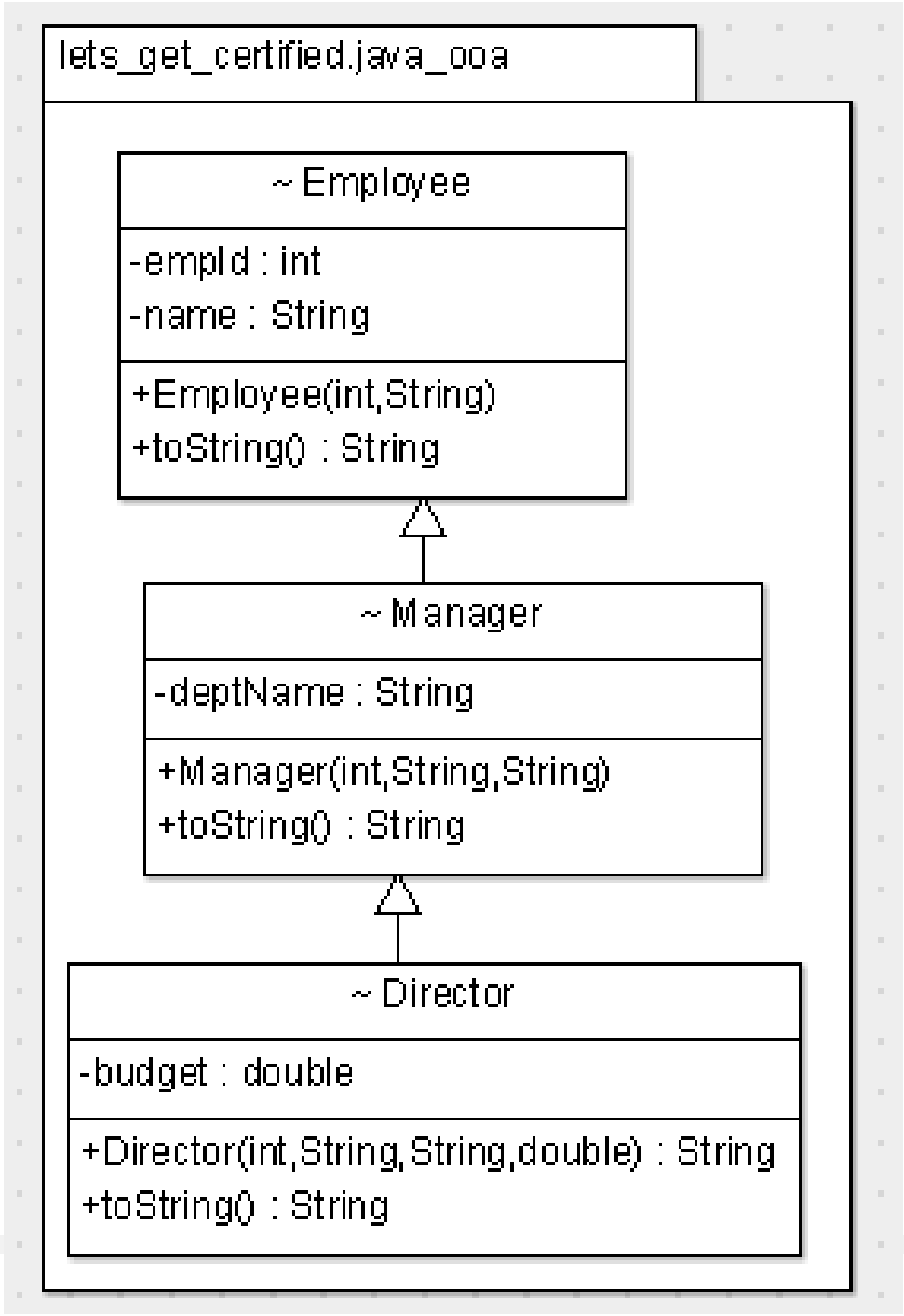
- Used to determine if a reference is referring to an object of a certain type (class or interface).
- Very useful when overriding *equals(Object)* from *Object*
- *public boolean equals(Object)*
  - the same as == i.e. are the references referring to the same object in memory?
  - override this to provide custom behaviour



super() and *super.*

- *super([args])* is how a constructor calls its parent constructor
- *super.XXX([args])* is how you call a parent method
- You cannot go super.super.method()

UML

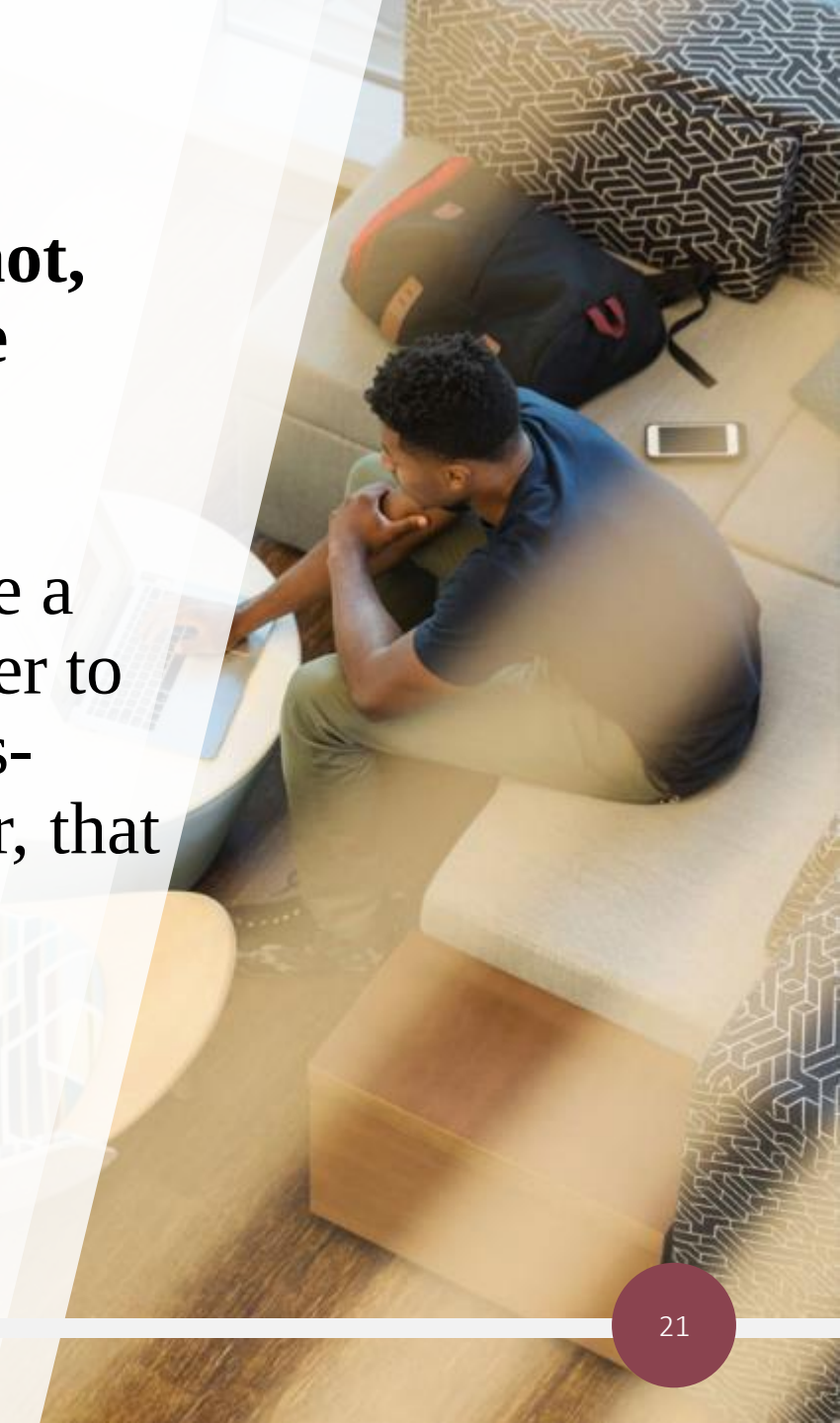

lets_get_certified.java_ooa
~ Employee
-empId : int
-name : String
+Employee(int,String)
+toString() : String
~ Manager
-deptName : String
+Manager(int,String,String)
+toString() : String
~ Director
-budget : double
+Director(int,String,String,double) : String
+toString() : String

protected

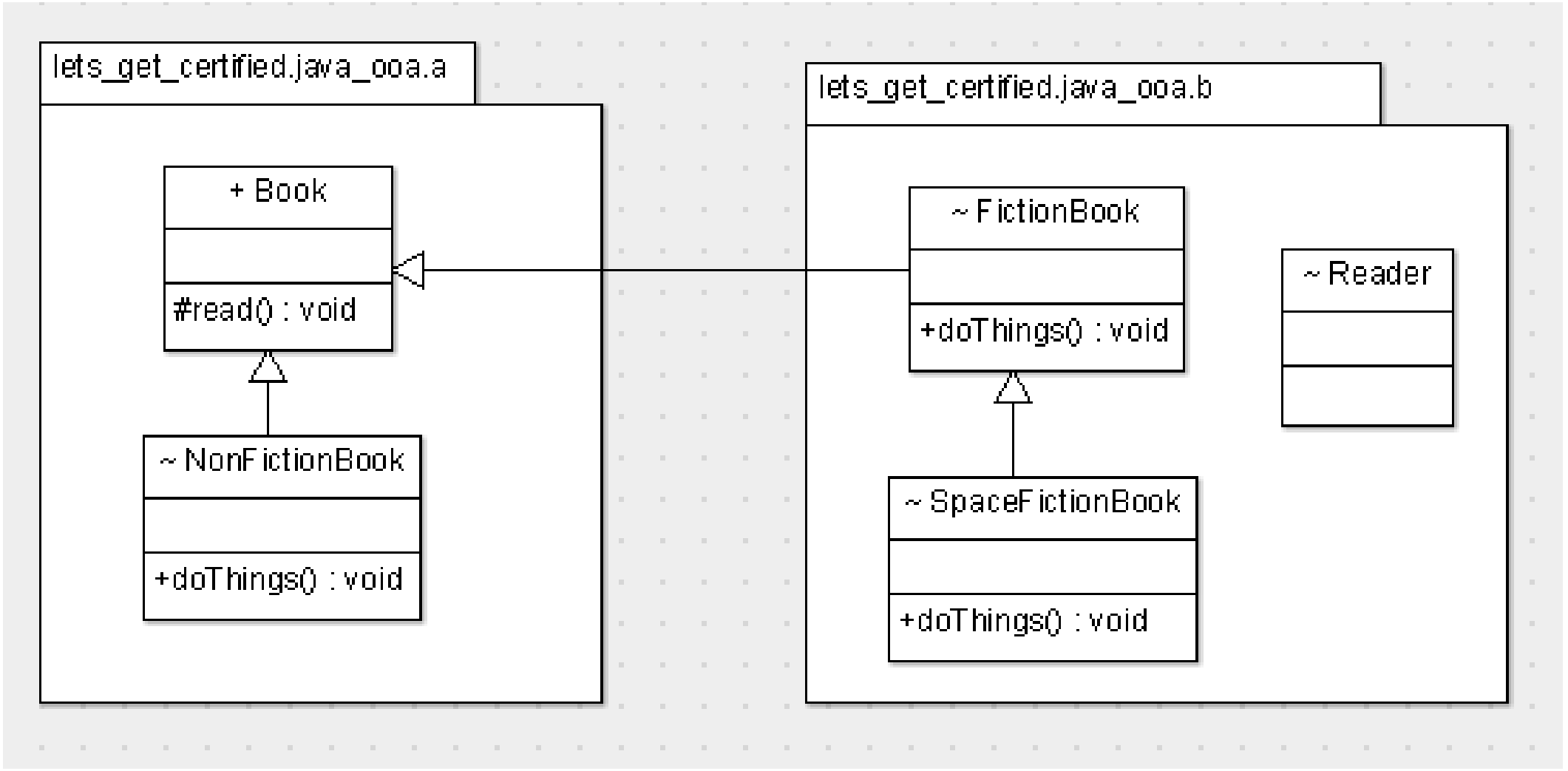
- Already discussed the access modifiers: *private*, *public* and package-private (no keyword).
- A *protected* member is similar to a package-private member in that a *protected* member may be accessed if the class accessing the member is in the same package
- The difference is a *protected* member can be accessed by a subclass (**via inheritance, in a very specific way**), even if the subclass is in a different package
- *protected* = package + kids

protected

- **A subclass-outside-the-package, believe it or not, <u>cannot</u> use a superclass reference to access the protected member it already has access to via inheritance!**
- In addition, a class outside the package <u>cannot</u> use a reference to the subclass-outside-the-package either to access the *protected* member i.e. once the subclass-outside-the-package inherits the *protected* member, that member becomes private to any code outside the subclass, with the exception of subclasses of the subclass.



UML

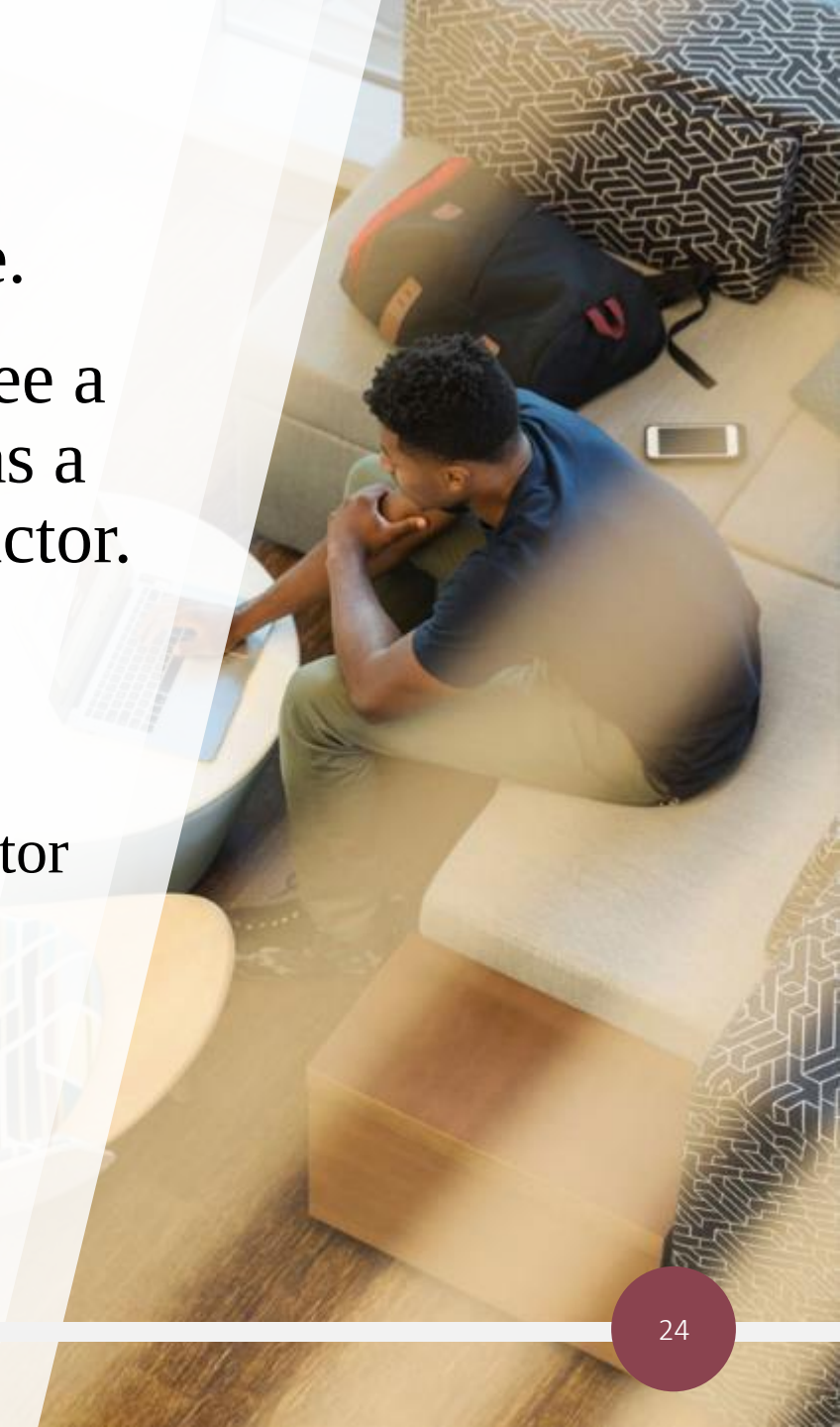

lets_get_certified.java_ooa.a
+ Book
#read() : void
~ NonFictionBook
+doThings() : void
lets_get_certified.java_ooa.b
~ FictionBook
+doThings() : void
~ SpaceFictionBook
+doThings() : void
~ Reader

Constructors

- Every class, including *abstract* classes, MUST have a constructor.
- Two key points about constructors:
  - constructors have exactly the same name as the class name
  - constructors have no return type.
- Constructors typically initialise the object state.
- Constructors are not inherited and as a result are never overridden but can be overloaded.

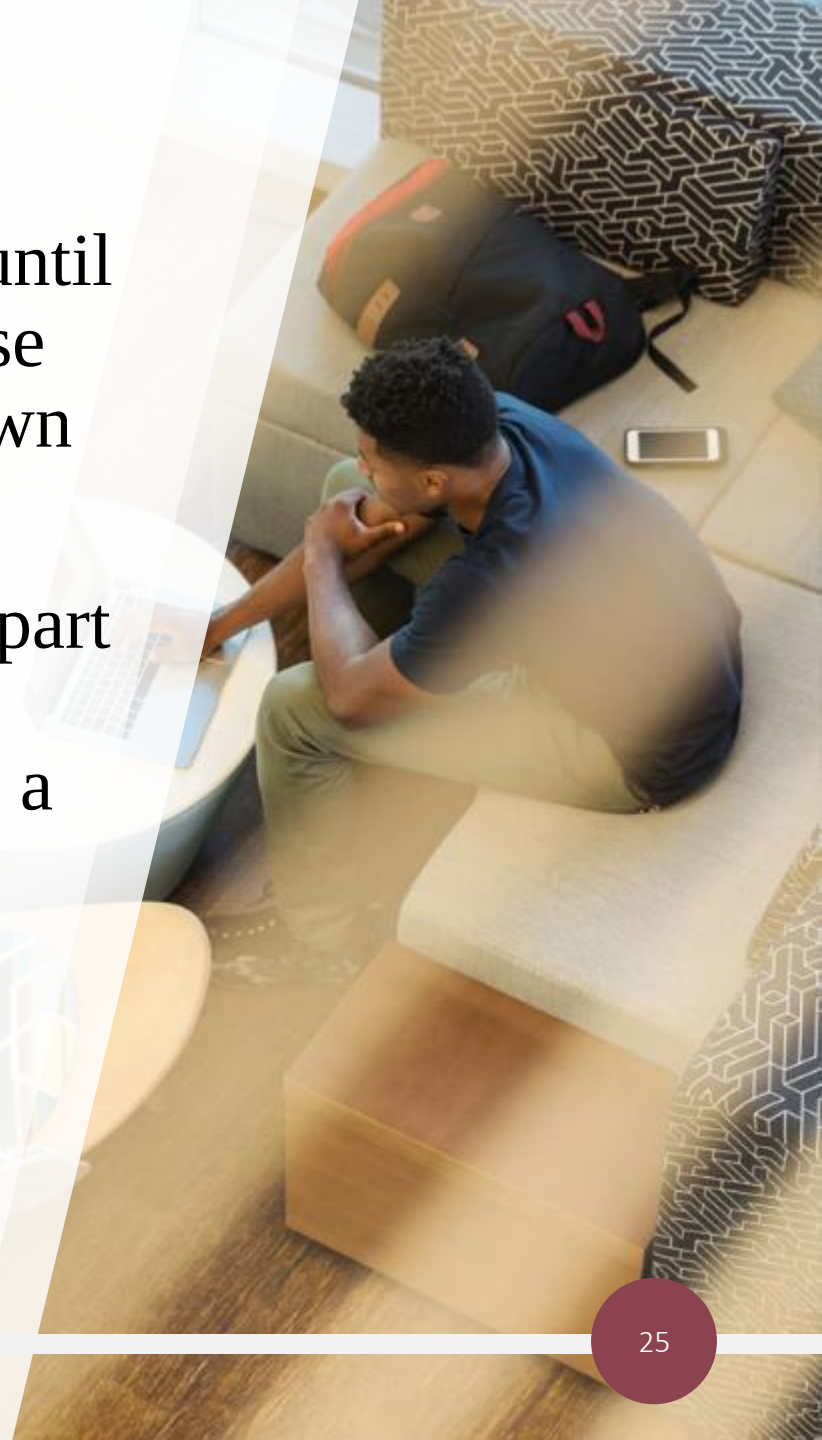
Constructors

- The constructor name must match the class name.
- Constructor must not have a return type; if you see a method with the same name as the class and it has a return type, then its a method and NOT a constructor.
- If you do not provide ANY constructor, a *default* constructor is provided by the compiler for you.
  - the *default* constructor is ALWAYS a no-arg constructor
  - it will have the same access as the class
  - it's first line will be "*super();*"



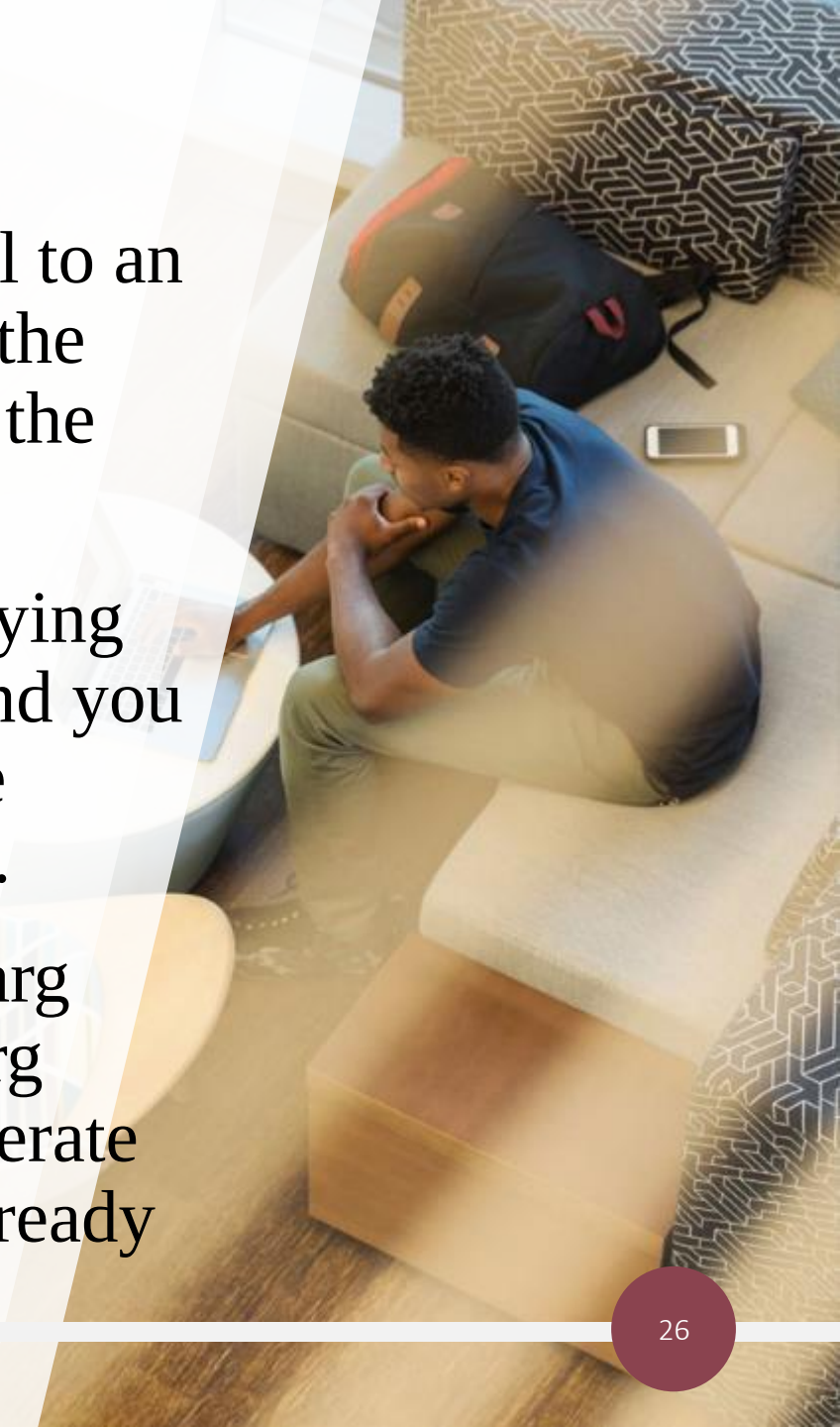
Constructors

- An instance variable/method cannot be accessed until after *super()* runs; this makes sense as you may use parent instance variables when initialising your own instance variables.
- Only *static* variables/methods can be accessed as part of the call to *this([args])* or *super([args])*
  e.g. *super(Animal.NAME)* is ok because *NAME* is a *static* variable.



Constructors

- Every constructor has, as its first statement, a call to an overloaded constructor (*this([args])*) or a call to the superclass constructor (*super([args])*). Note that the compiler can (and often does) insert *super()*.
- If you do type in a constructor (as opposed to relying on the compiler-generated *default* constructor) and you do NOT type in *super([args])* or *this([args])*, the compiler will insert *super()* as the first statement.
- Although the default constructor is always a no-arg constructor, you are free to insert your own no-arg constructor (because the compiler will never generate the default constructor if there is a constructor already coded).



Compiler Generated Constructor Code

`class A {}`	`class A{`
	`  A() {`
	`  super();`
	`  }`
	`}`

`class A {`
`  A(){}`
`}`
	`class A{`
	`  A() {`
	`  super();`
	`  }`
	`}`

Compiler Generated Constructor Code

`public class A {}`	`public class A{`
	`  public A() {`
	`  super();`
	`  }`
	`}`

`class A {`
`  A(String s){}`
`}`
	`class A{`
	`  A(String s) {`
	`  super();`
	`  }`
	`}`

Compiler Generated Constructor Code

`class A {`
`  A(String s){`
`  super();`
`  }`
`}`
	Nothing.

`class A {`
`  void A(){}`
`}`
	`class A{`
	`  void A(){}`
	`  **A() {**`
	`  **super();**`
	`  **}**`
	`}`

Compiler Generated Constructor Code

```
// No default constructor
// generated by the compiler
// as there are constructors
// already in the class.
class Horse{
    Horse(){}
    Horse(String name){}
}
```

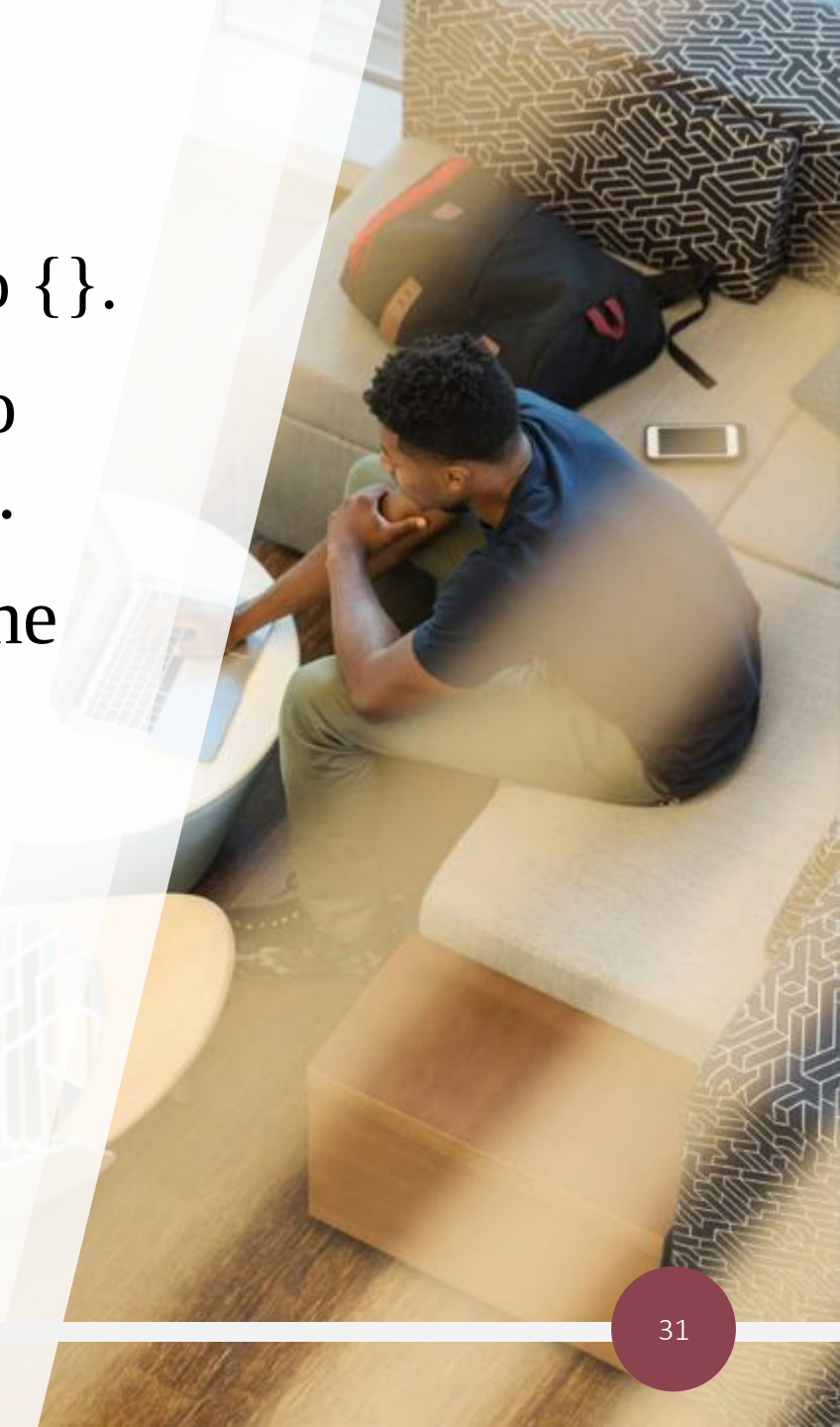
```
// No default constructor
// generated by the compiler
// as there is one constructor
// already in the class.
class Horse{
    Horse(String name){}
}
```

```
// The default constructor IS
// generated for both these
// classes.
class Cow{} // no ctor at all
class Pig{
    void Pig(){} // method!
}
```

```
class Clothing{
    Clothing(String type){}
}
class Shirt extends Clothing{
    // The problem here is that the compiler inserts:
    //    Shirt(){
    //       super();
    //    }
    // but there is no no-arg constructor in Clothing!
}
```

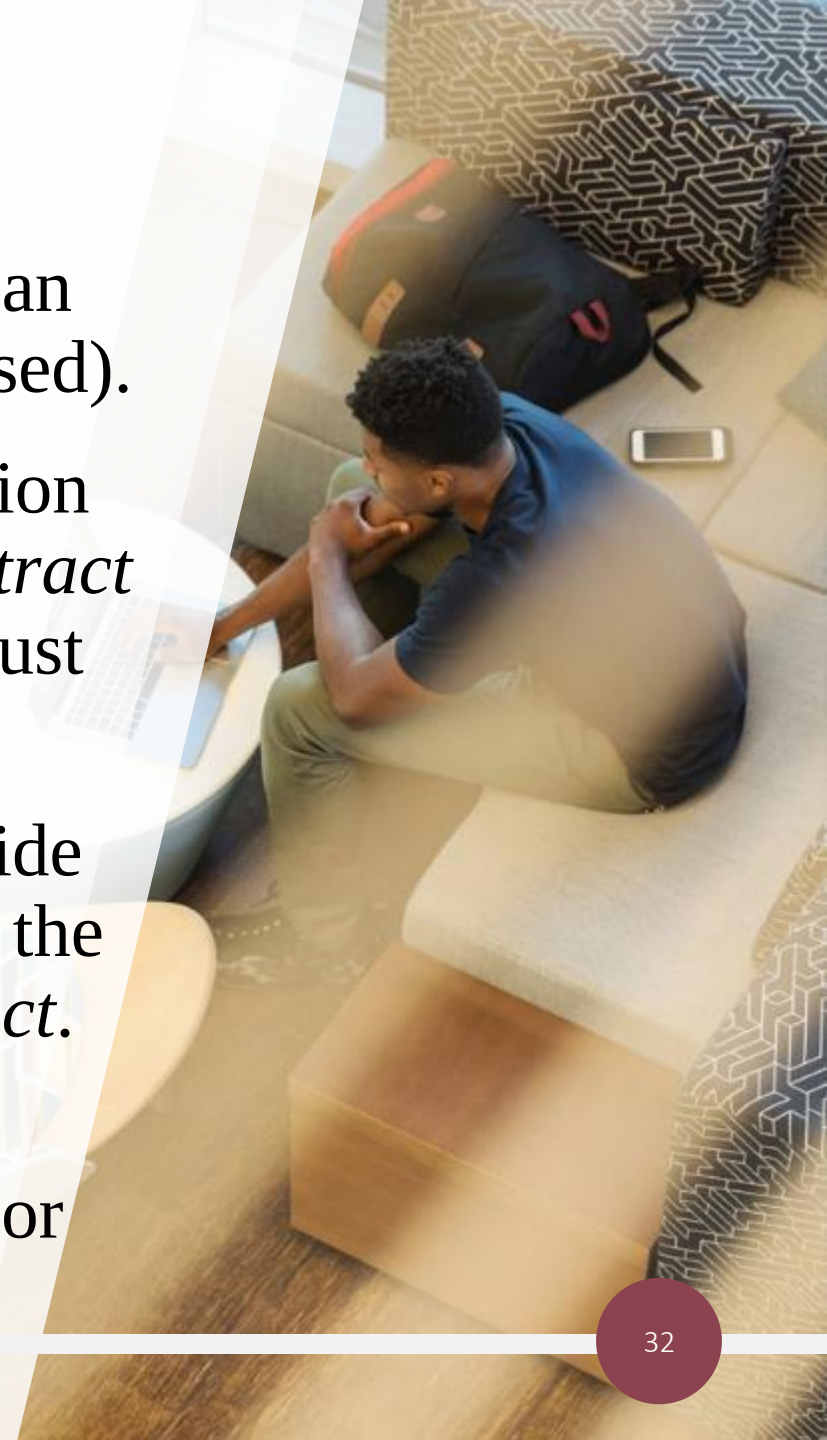
abstract methods

- An *abstract* method has no implementation i.e. no {}.
- You mark a method *abstract* when you want to force subclasses to provide an implementation.
- If you have 1 or more *abstract* methods then the class **must** be *abstract*.
- *abstract* methods must never be marked:
  - *final* (do not override) because *abstract* means please override
  - *private* (not inherited, no way to override)



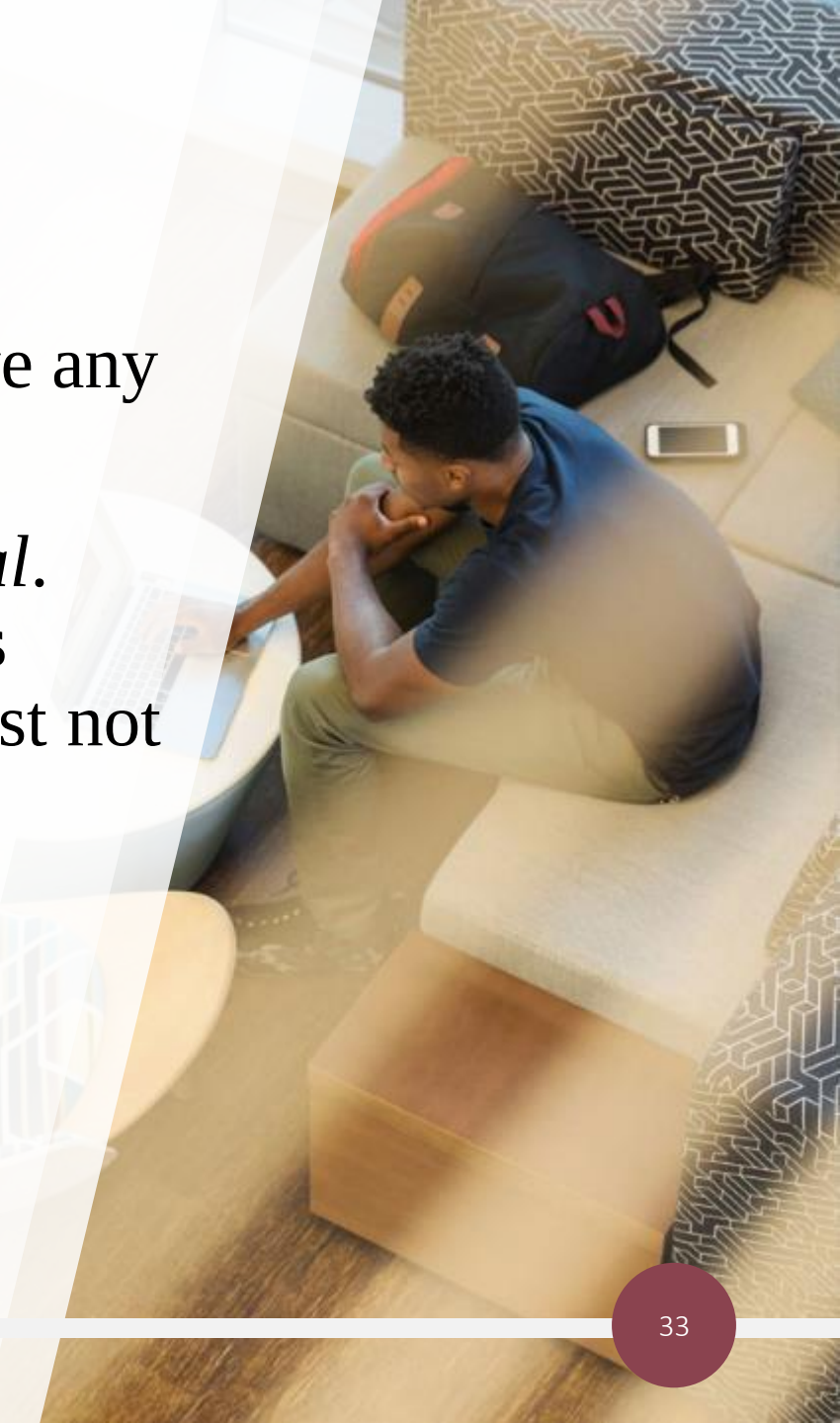
abstract classes

- An *abstract* class cannot be instantiated (*new*'ed); an *abstract* class's purpose is to be extended (subclassed).
- If a class has an *abstract* method(s) then the intention if for the *abstract* class to be extended and the *abstract* method(s) overridden (or else the subclass itself must be *abstract*)
- Any class that *extends* an *abstract* class must provide implementations for all of the *abstract* methods of the superclass, unless the subclass itself, is also *abstract*.
  - the rule is: the first concrete (non-abstract) subclass of an *abstract* class must implement (or inherit) all *abstract* methods of the superclass.



abstract classes

- An *abstract* class can contain 0 or more *abstract* methods i.e. an *abstract* class does not need to have any *abstract* methods.
- A class cannot be marked as both *abstract* and *final*. They have opposite meanings - an *abstract* class is intended to be subclassed whereas a *final* class must not be subclassed

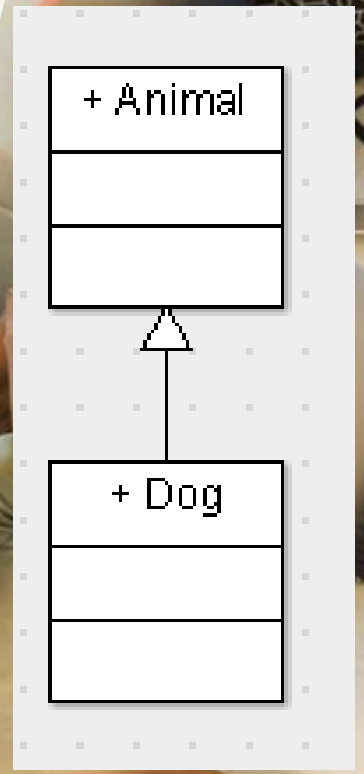


Upcasting and Downcasting

- Given the hierarchy on the right:

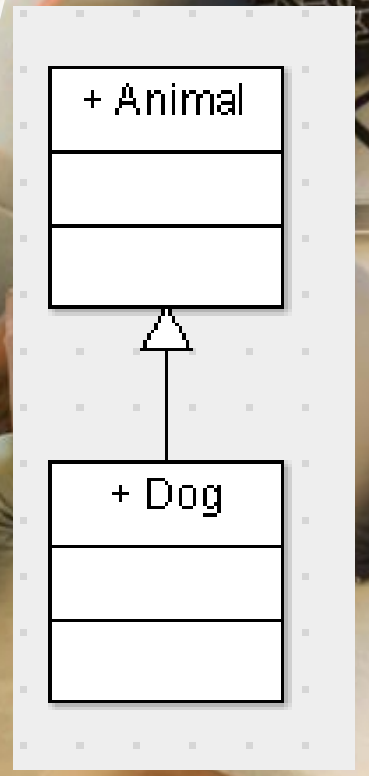
  Animal ad = new Dog(); // upcasting

- Upcasting happens implicitly (as above) because every Dog “is-a” Animal. This will never cause a problem because the Dog class will, via inheritance, have all the methods that you could possible call using the Animal reference.
- Remember, the reference type determines the methods you can call.
- Upcasting is so called because, reading the line of code above from right to left, we are going “up” the inheritance hierarchy from Dog to Animal.



Upcasting and Downcasting

- What if you want to call a Dog method that is not in the Animal class and you "only have" an Animal reference?
- This is where downcasting comes in. Remember, that a reference must never point "up" the hierarchy e.g. a Dog reference must never refer to an Animal object because Dog may well have extra methods that Animal does not. If this occurs, you get a *ClassCastException*.
- *instanceof*, as we saw in *equals()*, can protect against this.

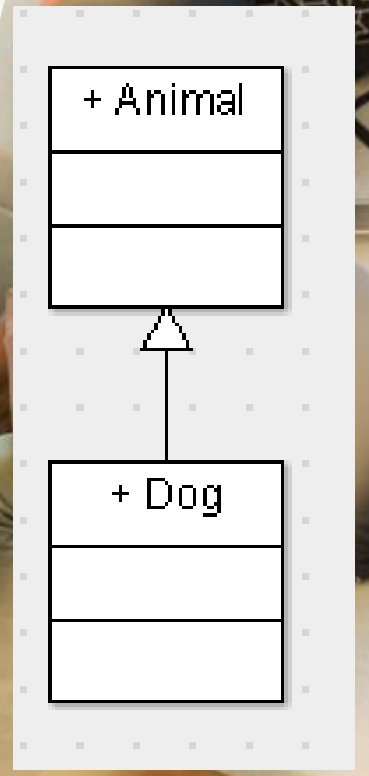


Upcasting and Downcasting

- The following line of code requires a (down)cast as the compiler realises it is potentially dangerous. The cast allows us to say to the compiler “I know what I am doing, go away!”

```
Dog d = (Dog) new Animal();// ClassCastException
```

- Downcasting is so called because, reading the line of code above from right to left, we are going “down” the inheritance hierarchy from Animal to Dog.



Upcasting and Downcasting