# O01 Classes and Objects 1

Class declaration
Object creation

# Creating Classes and Objects

The following slides describe the *mechanics* of creating a class and creating objects (instances of that class) in Java.

Some of the mechanics *will not make much sense* until later when the relevant concepts are explained. For now, treat these as boilerplate (stuff you ‘just do’).

# Class Declaration

A class declaration will have the following, in order:

- Any **modifiers** (`public`, `private`, etc.)
- The keyword `class`
- The **class' name** (first letter capitalized)
- Optional **superclass' name** preceded by `extends`
- Optional list of **interfaces** preceded by `implements`
- The class **body** surrounded by braces {}

# Member Variable Declaration

Three kinds:

- Class and instance variables, called **fields**
- Variables within a method, called **local variables**
- Method arguments, called **parameters**

Member variables will have the following, in order:

- Any **modifiers** (`public`, `private`, etc.)
- The field's **type**
- The field's **name**

# Constructors

A constructor is a special method that is automatically executed when an instance is created.

Constructors differ from normal methods:

- They have **no return type**.
- They have the **same name as the class**.

If no constructor is provided, the compiler will automatically call the constructor for the class' superclass

# Creating Objects

A statement creating an object has three parts:

- **Declaration** (a referring variable and type)
- **Instantiation** (the `new` keyword) – new object on heap
- **Initialization** (call to constructor) – initialise object

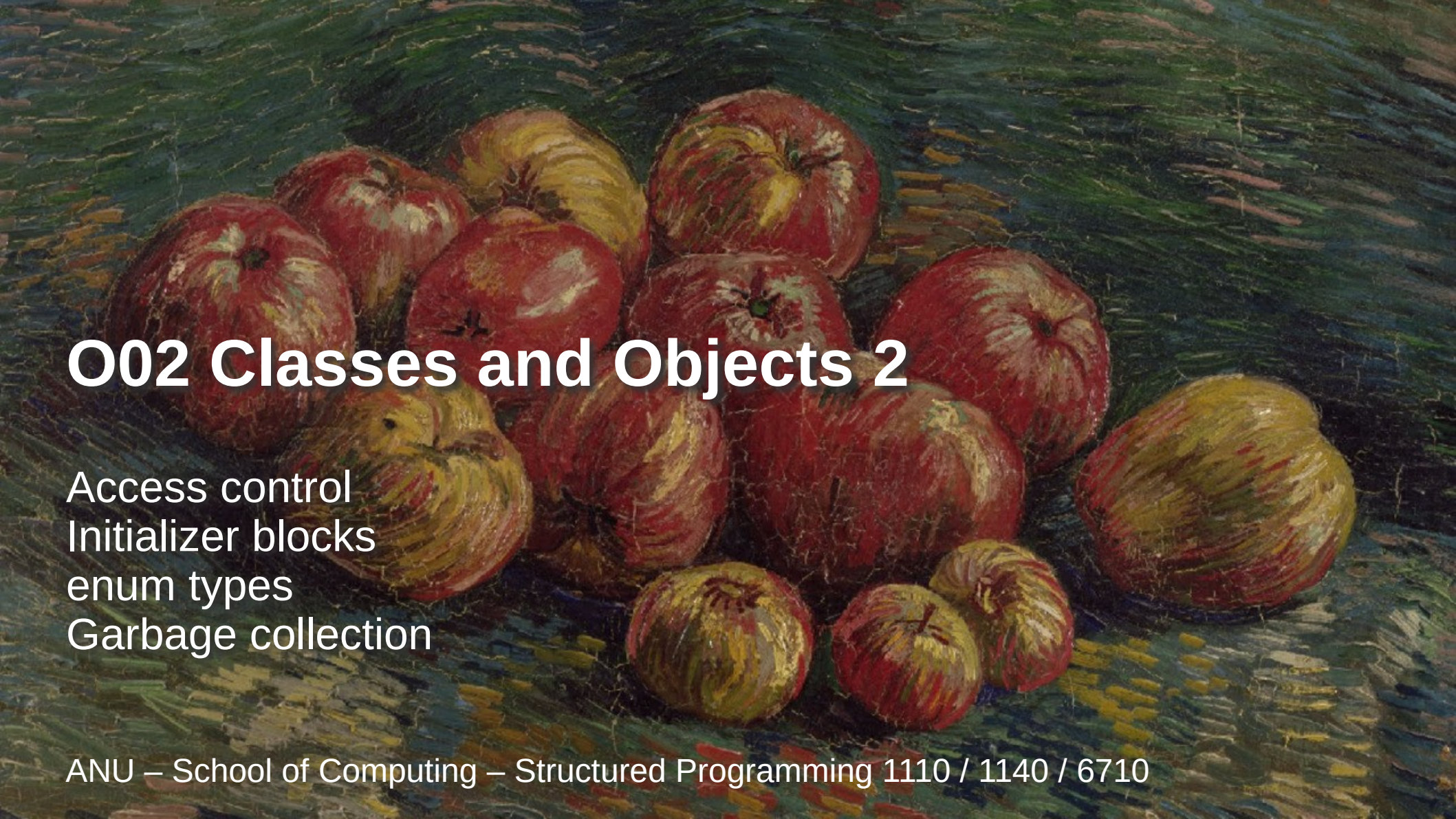
# Using Objects

Outside a class, an object reference followed by the dot '.' operator must be used:

- Reference the object's fields
  - Object reference, '.', field name
- Call the object's methods
  - Object reference, '.', method name, arguments in parentheses

Within instance methods, the object's fields and methods can be accessed directly by name, (optionally with the `this` keyword).

  - `fieldName` or `methodName()`
  - `this.fieldName` or `this.methodName()`



# O02 Classes and Objects 2

Access control
Initializer blocks
enum types
Garbage collection

# Variable Scope

- **Scope -** where  in your code a variable can be accessed
  - Scope of local variables / parameters limited to containing method / block. Disappear once a method returns (stack frame is popped).
  - Scope of class fields (`static` qualifier) and instance fields depend on the access control modifiers (`private`, `public`, etc).

# Access Control

Access modifiers determine whether fields and methods may be accessed by other classes

- Top level: `public` or package-private
- Member level: `public`, `protected`, package-private, or `private`

| Modifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| `public` | ✓ | ✓ | ✓ | ✓ |
| `protected` | ✓ | ✓ | ✓ | ✗ |
| *no modifier* | ✓ | ✓ | ✗ | ✗ |
| `private` | ✓ | ✗ | ✗ | ✗ |

# Class Members

The `static` keyword identifies class variables, class methods and constants.

- A **class variable** is common to all objects (there is only one version)
- A **class method** is invoked using a class name (not an object reference) and executes independently of any particular object.
- A **constant** can be declared by combining the `final` modifier with the `static` keyword.

# The `this` keyword

Within instance methods and constructors, the `this` keyword refers to the object whose method or constructor is being called.

- Disambiguating field names from parameters
  - Parameters and instance field names may clash. The `this` keyword explicitly refers to the instance.
- Calling other constructors
  - When there are multiple constructors, they may call each other using `this` as if it were the method name.

# Initializer Blocks

Fields may be initialized when they are declared.  They can also be initialized by **initializer blocks**, which can initialize fields using arbitrarily complex code (error handling, loops, etc.).

- A **static initializer** block is consists of code enclosed by braces ‘{}’and preceded by the `static` keyword. It runs when the class is first accessed.
- A **instance initializer** block does not have the `static` keyword, and runs before the constructor body of the class.

# Enum Types

An **enumerated type** is defined with the `enum` keyword.
A variable of enum type must be one of a set of predefined values.
This is useful for defining non-numerical sets such as `NORTH`, `SOUTH`, `EAST`, `WEST`, or `HD`, `D`, `CR`, `P`, `N`, etc.

- May have other **fields**
- May have **methods**
- May use **constructors**
- Can be used as argument to **iterators**

# Garbage Collection

In some object oriented languages, the programmer must keep track of objects and delete them when they are no longer used.

This is error-prone.

Java uses a garbage collector to automatically collect objects that can no longer be used. Garbage collection approximates *liveness* by reachability (the collector conservatively assumes that any reachable object is live).

# O03 Interfaces

Interfaces

# Interfaces

An `interface` can be thought of as a contract that a class can satisfy.

- Uses `interface` keyword rather than `class`
- Cannot be instantiated (can't be created with `new`)
- Can contain (all implicitly `public`):
  - *Abstract methods* (method declaration without a body)
  - *Default methods* (using `default` modifier)
  - Static methods (using `static` modifier)
  - Constants (implicitly `static final`)
- Classes implement interfaces via `implements` keyword
  - A class which implements an interface must provide the specified functionality.

# Interfaces as Types

An interface can be used as a type

- A variable declared with an interface type can hold a reference to a object of any class that implements that interface.

# O04 Inheritance 1

Inheritance
Hiding and overriding
Polymorphism
The super keyword

# Inheritance

A class that inherits is known as a *subclass*, *derived class*, or *child class*. Its parent is known as a *superclass*, *base class*, or *parent class*.

- Subclasses inherit via the `extends` keyword
- All classes implicitly inherit from `java.lang.Object`

# Overriding and Hiding Methods

- Instance methods
  - If method has same signature as one in its superclass, it is said to **override**. Mark with `@Override` annotation.
  - Same name, number and type of parameters, and return type as overridden parent method.
  - The type of the instance (not the variable referencing it) determines the method.
- Class methods
  - If it has same signature, it **hides** the superclass method.
  - The class with respect to which the call is made determines the method.

# Polymorphism: “Many-forms”

A reference variable may refer to an instance that has a more specific type than the variable.

The method that is called depends on the type of the instance, not the type of the reference variable.

This overriding of methods is a form of **runtime polymorphism** (actual underlying type will dynamically determine the behaviour). Interfaces also provide a form of runtime polymorphism.

Method overloading (same name, different type signatures) and operator overloading (e.g., +) are a form of **compile-time polymorphism**.

# Hiding Fields

When a subclass uses a field name that is already used by a field in the superclass, the superclass' field is **hidden** from the subclass.

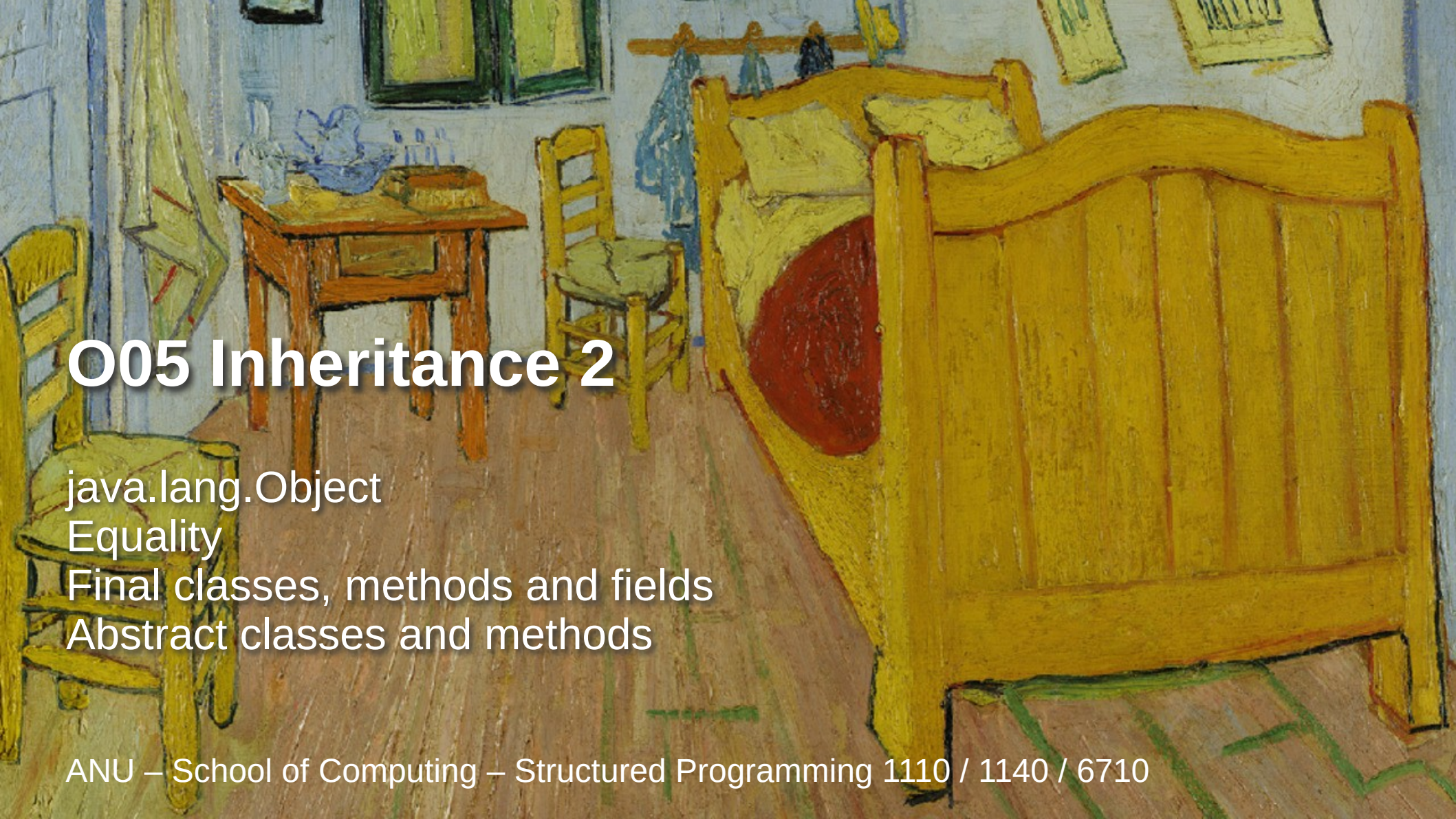
Hiding fields is a bad idea, but you can do it.

# The `super` keyword

You can access overridden (or hidden) **members** of a superclass by using the `super` keyword to explicitly refer to the superclass.

You can call superclass constructors by using `super`() passing arguments as necessary.





# O05 Inheritance 2

java.lang.Object
Equality
Final classes, methods and fields
Abstract classes and methods

# Object as superclass

In Java all classes ultimately inherit from **one** root class: `java.lang.Object`. Implemented methods:

- `clone()` returns copy of object
- `equals(Object obj)` establishes equivalence
- `finalize()` called by GC before reclaiming
- `getClass()` returns runtime class of the object
- `hashCode()` returns a hash code for the object
- `toString()` returns string representation of object

# Note on Equality

- Variables for **primitive types**:
  - Use == for equality.
  - Have no methods (i.e. have no `equals()`).
- Variables that **reference objects**:
  - `a == b`: true iff a and b reference the **same object instance**.
    - Checking the variable's immediate value is the same, which is a reference.
    - Two different instances can have exactly the same fields, and yet not be ==.
  - `a.equals(b)`: class-specific (semantic) object equality.
    - Default inherited from `java.lang.Object` is just ==.

# Final Classes and Methods

The `final` keyword in a class declaration states that the class **cannot** be subclassed.

The `final` keyword in a method declaration states that the method **cannot** be overridden.

# Abstract Classes and Methods

The `abstract` keyword in a class declaration states that the class is abstract, and therefore cannot be instantiated (its subclasses may be, if they are not abstract).

The `abstract` keyword in a method declaration states that the method declaration is abstract; the implementation must be provided by a subclass (like abstract methods in an interface, but here we need to be explicit and use the keyword).