O01 Classes and Objects 1

Class declaration
Object creation

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Classes and objects

Java is an *object-oriented* language.

- Objects combine state (fields) and behaviour (methods).
- A class defines a type objects (what fields and methods they have).
  - Each objects is an instance of a class.
- Classes form a hierarchy.
  - `java.lang.Object` is the root (ultimate ancestor) class of all Java classes.

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 {}

Class Member Declarations

Fields and methods of a class are known as “class members”.

Field (member variable) declarations have the following, in order:

- Any **modifiers** (`public`, `private`, `static`, etc.)
- The field’s **type**
- The field’s **name**
- (optional) a ‘=’, followed by an initial value expression.

Declarations are statements – end with ‘`;`’.

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 defined, the compiler will automatically call the constructor for the class' superclass

Note: If no other constructor defined, class inherits a no-parameter constructor from `Object`.

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.

Creating Objects

An object-creating expression consists of

- the keyword `new`
- followed by a call to the class’ constructor

Typically, the newly created object is assigned to a variable of matching type (class).

Objects may be deleted automatically when they are known to no longer be in use (garbage collection).

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

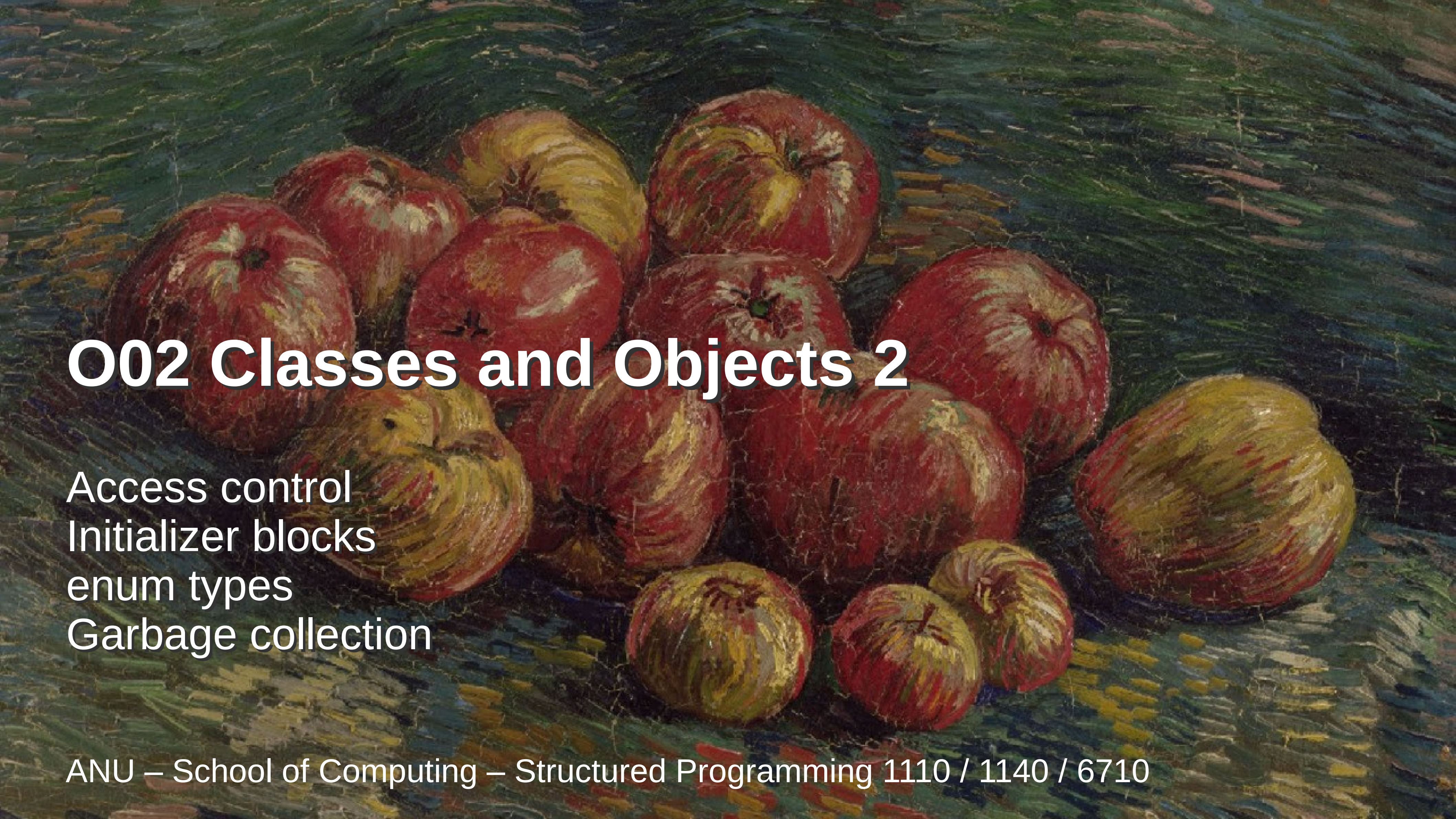
Within instance methods, the object’s fields and methods can be accessed directly by name, (optionally with the `this` keyword).

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

Overloading

A class can have several methods with the same name, but different arguments (number, type, order), often called “overloading”.

- Overloaded methods may have different return types.
- You can overload the constructor.



O02 Classes and Objects 2

Access control
Initializer blocks
enum types
Garbage collection

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Variable Scope

The **scope** of a variable is the section of code from within which it can be accessed.

- The scope of local variables and parameters is limited to the containing method or block.
  - Local variables cease to exist when execution leaves the method or block.
- The scope of class and instance fields depends on the access control modifiers (`private`, `public`, etc).

Access Control

Access modifiers determine which other classes can access fields and methods:

- Top-level: `public` or package-private (no modifier).
- Member level: `public`, `protected`, package-private, or `private`

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

Class Members

The `static` modifier keyword identifies class variables and methods.

- A **class variable** is shared by all instances of the class.
- A **class method** is called without reference to an object
  - Cannot use `this` in a class method (there is no “this”).
  - A class method can only reference class fields.
  - Class methods can be referenced (called) from outside the class using the class 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**
  - use static `values()` method.

O03 Interfaces

Interfaces
Abstract classes and methods

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

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.

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 applied selectively and explicitly).

O04 Inheritance

Inheritance
Hiding and overriding
Polymorphism
The super keyword

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

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 modifiers, return type, name, and sequence of parameter types as the overridden parent method.
  - **Dynamic dispatch**: The type of the object (not the variable referring to it) determines which method is called.
- 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**.

The Object superclass

All Java classes ultimately inherit from **one** root class: `java.lang.Object`. Some of its methods are:

- `clone()` returns (shallow) copy of object
  - Note: cloning is not automatically supported by all classes.
- `equals(Object other)` establishes semantic 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

The `super` keyword

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

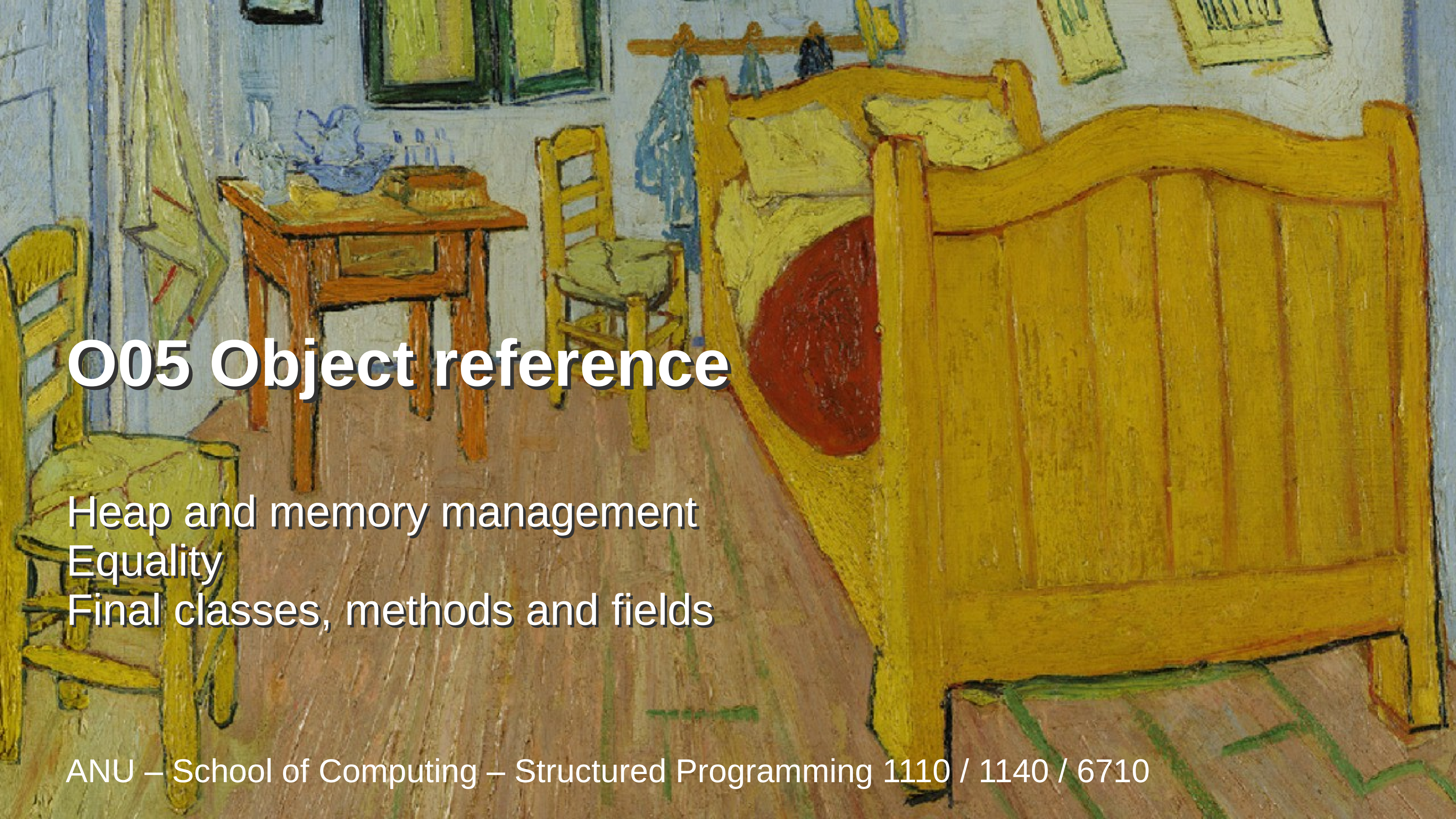
You can call superclass constructors by using `super`() passing arguments as necessary.

Type Casting

A reference to an object of a given class can be explicitly converted to a reference to a subclass: this is called (dynamically) “type casting”.

Because it is not guaranteed that the object is of the subclass, explicit casting can always result in a `ClassCastException`, which must be caught.

```
Try {
  SubClass y = (SubClass)x;
catch (ClassCastException e) {
  // statements to execute if x is not of class SubClass
}
```



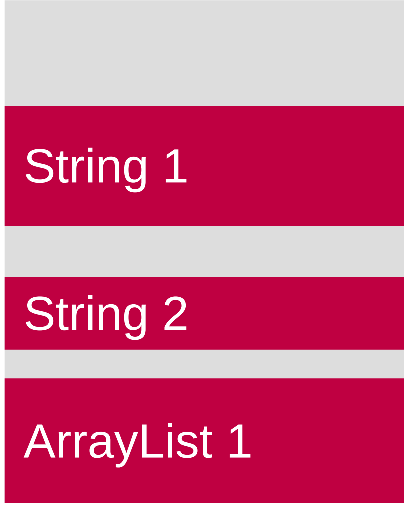
O05 Object reference

Heap and memory management
Equality
Final classes, methods and fields

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

The Heap

The heap: a large region(s) of memory used to store dynamically allocated objects (objects created with `new`).



Variables and References

- For variables of **primitive types**, the value is stored directly.
- For variables of **reference types (all objects)**, the “value” stored is a *reference* to an object stored on the heap.
  - Such variables can be set to `null` (reference to nothing).
  - Method calls, fields automatically access the object pointed to.
    - `NullPointerException` thrown if reference is `null`
  - More than one variable can *refer to the same object*.

Equality

- Variables of **primitive types**:
  - Use == for equality.
  - Have no methods (i.e. have no `equals()`).
- Variables that **reference objects**:
  - `a == b`: true iff a and b refer to 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 ==.

Garbage Collection

In Java, there is no explicit deallocation of objects.

A *garbage collector* automatically reclaims heap space used by objects that are no longer reachable (no longer referenced, directly or indirectly, by any variable in the program).

The `final` modifier

- A `final` field can not be reassigned
- A `final` method cannot be overidden
- A `final` class cannot be subclassed.

A `static final` field of a primitive type is like a constant.

A `static final` field of a reference type will always refer to the same object, but that object may change.