Australian National University

Classes and Objects 1

01

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)
- Initialization (call to constructor)

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()`

Classes and Objects 2

02

Locals, globals, heap
Garbage collection
Initializers, access control
enum types

Locals (stack), Globals (statics), and Heap (objects)

Local variables are declared within the scope of a method and hold temporary state. They disappear once the method returns.

Global variables (a.k.a. '*class variables*') are declared within the scope of a class (with a `static` qualifier), and exist as long as the class is loaded (which is usually for the duration of the program).

Heap variables (a.k.a. *'instance variables'*) are declared within the scope of a class (without a `static` qualifier), and exist as long as the containing instance is reachable.

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).

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.

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 and Instance 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.

Initializers

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**


Australian
National
University
Interfaces
03
Structured Programming 1110/1140/6710

Interfaces

An interface can be thought of as a contract.
A class which implements an interface *must* provide the specified functionality. Compared to a class, an interface:

- Uses `interface` keyword rather than `class`
- Cannot be instantiated (can't be created with `new`)
- Can *only* contain constants, method signatures (not the bodies), nested types
  - (Java 8+ allows `default` and `static` methods)
- Classes implement interfaces via `implements` keyword

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.

Inheritance 1

04

Inheritance
Overriding and hiding
Polymorphisim
The super keyword

Inheritance

An inherited class 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** 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

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.

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.

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

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

Australian National University

Inheritance 2

05

java.lang.Object
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*

Final Classes and Methods

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

The `final` keyword in a method declaration states that the method *may not* 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.