

OO Concepts in Java

- Classes define the characteristics of instances and bring together in one place both **state** (attributes) and **behavior** (methods). This is called **encapsulation**.
- The state of instances is kept **private** to the instance and only accessed or modified through **public** methods. This is called **information hiding**.

OO Concepts in Java (2)

- The state of an instance is accessed or modified by calling a **public method** defined within the instances class. This is referred to as **message passing**.
- The set of public methods defined in a class is called the class's **interface** to its clients.

OO Concepts in Java (3)

- Any number of instances may simultaneously exist and each has its own private and **independent** state.
- Instances are created by calling the **constructor** method, which both **creates** and **initializes** the instance. A default constructor is provided in a class unless a user defines one (**or more**).

OO Concepts in Java (4)

- Any number of constructors may be defined so long as they all have different numbers and / or types of parameters. The same applies to other identically named methods within a class. This is called **overloading**.

Packages

- Java programs consist of one or more packages.
- Packages contain one or more classes.
- Every source file in Java is part of a package.
- A source file can begin with a ***package*** statement, which indicates what package the source file **belongs to**.

Packages (2)

- If a source file does not begin with a ***package Packagename;*** statement, the file is assumed to be part of the **default**, unnamed package, which includes the compiled classes in the current directory.
- The ***import java.awt.*;*** statement is how to have the java compiler import all the source files in the subdirectory ***awt*** which is in the directory ***java***.

Packages (3)

Helps manage software complexity:

- Separate namespace for each package – package name may added in front of actual name.
- Put generic / utility classes in packages – avoid code duplication.

Package – Import

- Import:
 - Make classes from package available for use.
 - Java API
 - java.* (core)
 - javax.* (optional)
- Example:

```
import java.util.Random;    // import single class
import java.util.*;        // all classes in package
...                        // class definitions
```


C++ Namespaces

- Namespaces are kind of like **packages** in Java.
- A mechanism for logically grouping declarations and definitions into a common declarative region.
- Reduces naming conflicts.

C++ Namespaces (2)

- The contents of the namespace can be accessed by code inside or outside the namespace:
 - Use the **scope resolution operator** to access elements from outside the namespace.
 - Alternatively, the **using** declaration allows the names of the elements to be used directly.

C++ Namespaces (3)

- Creating a namespace:

```
namespace smallNamespace  
{  
    int count = 0;  
    void abc();  
} // end smallNamespace
```

- Using a namespace:

```
using namespace smallNamespace;  
count += 1;  
abc();
```

C++ Namespaces (4)

- Items declared in the C++ Standard Library are declared in the *std* namespace.
- C++ *include* files for several functions are in the *std* namespace:
 - to include input and output functions from the C++ library, write

```
#include <iostream>
```

```
using namespace std;
```
- Any standard C routines (*malloc*, *printf*, etc.) are defined in the global namespace because **C** does not have namespaces.

using namespace

```
#include <iostream>
...
std::string question =
    "How do I list directory content?";
std::cout << question << std::endl;

using namespace std;

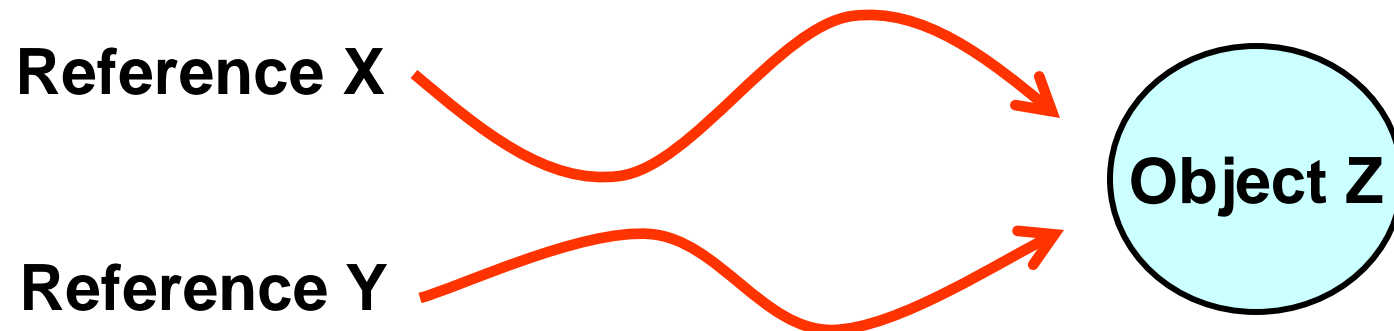
string answer = "Type ls.";
cout << answer << endl;
```

Classes

- A **class** is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind.
- An **object** is an instance of a class.

References & Aliases

- Reference:
 - A way to get to an object, not the object itself.
 - All variables in Java are **references** to objects.
- Alias:
 - Multiple references to same object.
 - “**X == Y**” operator tests for alias.
 - **X.equals(Y)** tests contents of object.



References & Aliases – Issues

- **Copying:**

- References

- `X = new Object();`

- `Y = X;` `// Y refers to same object as X`

- Objects

- `X = new Object();`

- `Y = X.clone();` `// Y refers to different object`
 `// (a duplicate)`

“this” Reference

- Description:
 - Reserved keyword.
 - Refers to object through which method was invoked.
 - Allows object to refer to itself.
 - Use to refer to instance variables of object.

“this” Reference – Example

```
class Node {  
    value val1;  
    value val2;  
  
    void foo(value val2) {  
        ... = val1;           // same as this.val1 (implicit this)  
        ... = val2;           // parameter to method  
        ... = this.val2;      // instance variable for object  
        bar( this );          // passes reference to object  
    }  
}
```

Inheritance

- Definition:
 - Relationship between classes when state and behavior of one class is a subset of another class.
- Terminology:
 - **Superclass** / parent (**base**) \Rightarrow More general class.
 - **Subclass** (**derived**) \Rightarrow More specialized class.
- Forms a class hierarchy.
- Helps promote code reuse.

- Subclass
 - = special case
 - = fewer entities
 - = more properties.

“super” Reference

- **Description:**

- Reserved keyword.
- Refers to superclass.
- Allows object to refer to methods / variables in superclass.

- **Examples:**

- `super.x` `// accesses variable x in superclass`
- `super()` `// invokes constructor in superclass`
- `super.foo()` `// invokes method foo() in superclass`

Class Variables

- Declared inside the class and outside the methods.
- The values of **class variable** may be inherited by subclasses and overridden.

Class Methods

- ***Class method***: a method which is inherited by all subclasses of the class it belongs to and which defines the behavior of the **collective** of instances rather than **individual** ones.
- An operation targeted to a class and not to an individual object. In C++, **class methods** are called **static member functions**.

Class Definition

```
class classname {  
    [ variable declaration; ]  
    [ method declaration; ]  
}
```

Methods:

```
[ class-modifiers ] return-type method-name  
    ( parameter list ) { [ body; ] }
```


Using Classes

- Declaring object variables:
classname variablename;
- Creating objects:
variable = new classname();
- Accessing variable members:
variable.v_member
- Accessing method members:
variable.method()

Deriving Classes

```
class classname extends parent-class {  
    [ variable declaration; ]  
    [ method declaration; ]  
}
```

- Inheritance
- Overriding methods

Constructors / Creation Methods

```
public void classname ( [ parameters ] )  
    { body; }
```

Constructor Methods

- Every class in Java has a least one constructor method.
- Constructor method has the same name as the class.
- Purpose of constructor is to perform any necessary initialization.
- The **return type** is implicitly an instance of the class.

Constructor Methods (cont)

- Can have **multiple** constructors if want to initialize an object in different ways.
- Implicit invokes constructor for superclass (if not explicitly included).

Initialization Block

- Definition:
 - Block of code used to initialize **static** & **instance** variables for class.
- Motivation:
 - Enable complex initializations for static variables.
 - Share code between **multiple** constructors for same class.

Initialization Block Types

- **Static initialization block:**
 - Code executed when class loaded.
- **Initialization block:**
 - Code executed when each object created (at beginning of call to constructor).

- Example:

```
class foo {  
    static { A = 1; }    // static initialization block  
    { A = 2; }           // initialization block  
}
```

Variable Initialization

- Variables may be initialized:
 - At time of declaration.
 - In initialization block.
 - In constructor.
- Order of initialization:
 1. Declaration, initialization block
(in the same order as in the class definition)
 2. Constructor.

Variable Initialization – Example

```
class Foo {  
    static { A = 1; }           // static initialization block  
    static int A = 2;           // static variable declaration  
    static { A = 3; }           // static initialization block  
    { B = 4; }                  // initialization block  
    private int B = 5;          // instance variable declaration  
    { B = 6; }                  // initialization block  
    Foo() {                     // constructor  
        A = 7;  
        B = 8;  
    }                           // now A = 7, B = 8  
}                               // initializations executed in order of number
```

Garbage Collection

- **Concepts:**
 - All interactions with objects occur through reference variables.
 - If no reference to object exists, object becomes **garbage** (useless, no longer affects program).
- **Garbage collection:**
 - Reclaiming memory used by un-referenced objects.
 - Periodically performed by Java.
 - Not guaranteed to occur.
 - Only needed if running low on memory.

Destructor

- **Description:**

- Method with name `finalize()`.
- Returns **void**.
- Contains action performed when object is freed.
- Invoked automatically by garbage collector
 - Not invoked if garbage collection does not occur.
- Usually needed only for non-Java methods.

- **Example:**

```
class foo {  
    void finalize() { ... }           // destructor for foo  
}
```

Overloading Methods

- Java supports method name overloading (borrowed from C++).
- Declared multiple methods with same name.

Overloading Methods (2)

- Each one can have a different type of parameter passed.
- Or each one can have a different number of arguments passed.
- Or each one can have the same arguments but in a different order.

Overloading Methods (3)

- Sources of overloading:
 - Constructors frequently overloaded.

- Example:

```
class foo {  
    foo() { ... }           // constructor for foo  
    foo(int n) { ... }      // 2nd constructor for foo  
}
```

Overriding Methods

- Create a subclass with the keyword “***extends***”.
- To override a method, the new method must have the same name, argument types, and return type as the method in the superclass.
- For example, say you want a window with a border.

Overriding Methods (cont)

- ***public class WindowWithBorder
extends Window {***
- Now the ***WindowWithBorder*** class inherits everything from the ***Window*** class.
- Then declare the new method.
private void drawBorder() {

Class Modifiers

- Access modifiers:
 - “***default***” (not explicitly specified),
public, private, protected.
- The ***static*** modifier.
- The ***final*** modifier.
- The ***synchronized*** modifier.
- The ***native*** modifier.

Modifier – Examples

```
public class foo {  
    private static int count;  
    private final int increment = 5;  
    protected void finalize { ... }  
}
```

```
public abstract class bar {  
    abstract int go() { ... }  
}
```

Modifying Methods

- ***Method modifiers*** in Java can be divided into two groups.
- Those that affect the scope of a method (**scope** refers to the region of a program that an item can be accessed).
- And those that do not affect the scope.

Scope

- **Scope:**

- Part of program where a variable may be referenced.
- Determined by location of variable declaration
 - Boundary usually demarcated by { }

- **Example:**

```
public MyMethod1() {  
    int myVar;  
    ...  
}
```



myVar accessible in
method between { }

Scope – Example

- **Example:**

```
package edu.ccu.cs ;  
public class MyClass1 {  
    public void MyMethod1() {  
        ...  
    }  
    public void MyMethod2() {  
        ...  
    }  
}  
public class MyClass2 {  
}
```

Method Method

Class

Class

Package

Scopes

Visibility Modifier

- Properties:
 - Controls access to class members.
 - Applied to instance variables & methods.
- Four types of access in Java:
 - Public
 - Protected
 - Package
 - Default if no modifier specified
 - Private

Most visible



Least visible

Visibility Modifier – Where Visible

- **“public”**
 - Referenced **anywhere** (i.e., **outside package**).
- **“protected”**
 - Referenced within package, or by **subclasses outside package**.
- **None specified (package)**
 - Referenced only within package.
- **“private”**
 - Referenced only **within** class definition.
 - Applicable to class fields & methods.

Visibility Modifier

- For **instance variables**:
 - Should usually be **private** to enforce encapsulation.
 - Sometimes may be **protected** for subclass access.
- For **methods**:
 - Public methods – provide services to clients.
 - Private methods – provide support other methods.
 - Protected methods – provide support for subclass.

Summary: Affect-the-Scope Modifiers

- **public**: the method can be accessed by any class.
- **private**: the method can be accessed only by methods within the same class.
- **Not explicitly specified**: the method can be accessed by methods in the class or methods in other classes in the same package.

Affect-the-Scope Modifiers (cont)

- **protected**: the method can be accessed by methods in the subclasses of the class.

Not-Affect-the-Scope Modifiers

- **final**: the method **cannot** be overridden by a method in a subclass.
- **static**: the method is a class method.
- **native**: the method body will be written in C and linked into the interpreter.
- **abstract**: the method is not defined in the class. It must be defined in a **subclass**.

Not-Affect-the-Scope Modifiers (cont)

- **synchronized**: the method will acquire a lock on the instance (or on the class, if it is a class method) before running and will relinquish the lock when it completes.

Modifier – Static

- Static variable:
 - Single copy for class.
 - Shared among all objects of class.
- Static method:
 - Can be invoked through **class name**.
 - Does not need to be invoked through object.
 - Can be used even if no objects of class exist.
 - Can not reference instance variables.

Modifier – Final

- Final variable:
 - Value can not be changed.
 - Must be **initialized** in every constructor.
 - Attempts to modify **final** are caught at compile time.
- Final static variable:
 - Used for constants.
 - Example:
final static int Increment = 5;

Modifier – Final (2)

- Final method:
 - Method **cannot be overridden** by subclass.
 - Private methods are implicitly **final**.
- Final class:
 - Class cannot be a superclass (extended).
 - Methods in final class are implicitly **final**.

Modifier – Final (3)

- Using **final** classes:
 - Prevents inheritance / polymorphism.
 - May be useful for
 - Security
 - Object oriented design
- Example – class **String** is **final**.
 - Programs can depend on properties specified in Java library API.
 - Prevents subclass that may bypass security restrictions.

Modifier – Abstract

- **Description:**
 - Represents generic concept.
 - Can not be instantiated.
- **Abstract class:**
 - Placeholder in class hierarchy.
 - Can be partial description of class.
 - Can contain non-abstract methods.
 - Required if any method in class is abstract.
- **Example:**

```
abstract class foo {           // abstract class
    abstract void bar() { ... } // abstract method
```

Interface

- **Description:**
 - Collection of
 - Constants
 - Abstract methods
 - Can not be instantiated.
- Classes can **implement** interface:
 - Must implement **all** methods in interface.
 - Example:
class foo implements bar { ... } // interface bar
- Similar to abstract class:
 - But class can “inherit” from **multiple** interfaces.