



University of Kelaniya

Kesavan Selvarajah

BSc (UOK), MSc (Reading, UCSC)

Temporary Lecturer

Software Engineering Department

Faculty of Computing and Technology

skesa231@kln.ac.lk

Object-Oriented Programming

CSCI 21052 / ETEC 21062



Packages

- **Packages** are containers for classes. They are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.



Packages

```
package mypackage; // This is the package declaration
```

```
public class List {  
    // Class implementation here  
}
```

```
package myotherpackage; // This is the package declaration
```

```
public class List {  
    // Class implementation here  
}
```



Data access control

- **Encapsulation** links data (variables) with the code (methods) that manipulates it.
- However, **Encapsulation** provides another important attribute: **Access control**.
- Through encapsulation, we can control what parts of a program can access the members of a class.
- A member access will be determined by the access modifier attached to its declaration.
- Java's access modifiers are **public**, **private**, and **protected**.



Data access control

```
public class MyClass {  
    public int publicVariable;  
  
    public void publicMethod() {  
        System.out.println("This is a public method.");  
    }  
}
```



Data access control

```
public class Main{
    public static void main(String[] args) {
        MyClass myObject = new MyClass();

        // Accessing a public variable from AnotherClass
        myObject.publicVariable = 42;
        System.out.println("value: " + myObject.publicVariable);

        // Calling a public method from AnotherClass
        myObject.publicMethod();
    }
}
```



Data access control

- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.

```
public class MyClass {  
    private int privateVariable;  
    public MyClass() {  
        this.privateVariable = 0; // Initialize the variable in the constructor  
    }  
    public void setPrivateVariable(int value) {  
        this.privateVariable = value;  
    }  
    public int getPrivateVariable() {  
        return privateVariable;  
    }  
}
```



Data access control

```
public class Main{  
    public static void main(String[] args) {  
        MyClass myObject = new MyClass();  
  
        // Attempt to access the private variable directly  
        myObject.privateVariable = 42; →  
        // Accessing the private variable using public methods  
        myObject.setPrivateVariable(42);  
        int value = myObject.getPrivateVariable();  
        System.out.println("Private variable value: " + value);  
    }  
}
```



Data access control

- In Java, the **protected** access modifier is used to restrict access to members (fields, methods, and nested classes) within the same class, subclasses (inheritance), and classes in the same package.
- This means that a protected member can be accessed within the class, its subclasses, and other classes in the same package.



Data access control

```
package mypackage; // This is the package declaration

public class MyClass {
    protected int protectedVariable; // This variable has protected access

    protected void protectedMethod() { // This method has protected access
        System.out.println("This is a protected method.");
    }
}
```



Data access control

```
package mypackage; // This is the package declaration

public class SubClass extends MyClass {
    public void accessProtectedMembers() {
        // Accessing protected members from a subclass
        protectedVariable = 42;
        System.out.println("Protected variable value: " + protectedVariable);
        protectedMethod();
    }
}
```



Data access control

```
package mypackage; // This is the package declaration

public class AnotherClass {
    public static void main(String[] args) {
        SubClass subObject = new SubClass();

        // Accessing protected members from a different class in the same package
        subObject.accessProtectedMembers();
    }
}
```



Data access control

```
public class Main{  
    public static void main(String[] args) {  
        SubClass subObject = new SubClass();  
  
        // Accessing protected members from a different package  
        subObject.accessProtectedMembers();  
    }  
}
```



Data access control

- When no access modifier is used, then by **default** the member of a class is public within its own package, but cannot be accessed outside of its package.

```
package mypackage; // This is the package declaration

public class MyClass {
    int packagePrivateVariable;

    void packagePrivateMethod() {
        System.out.println("This is a package-private method.");
    }
}
```



Data access control

```
package mypackage; // This is the package declaration

public class AnotherClass {
    public static void main(String[] args) {
        MyClass myObject = new MyClass();

        // Accessing a package-private variable and method within the same package
        myObject.packagePrivateVariable = 42;
        System.out.println("value: " + myObject.packagePrivateVariable);
        myObject.packagePrivateMethod();
    }
}
```



Data access control

```
public class Main{  
    public static void main(String[] args) {  
        MyClass myObject = new MyClass();  
  
        // Accessing a package-private variable and method from outside the package  
        myObject.packagePrivateVariable = 42;  
        System.out.println("value: " + myObject.packagePrivateVariable);  
        myObject.packagePrivateMethod();  
    }  
}
```



Static members

- There will be times when we want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static**.



Static members

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare both methods and variables to be static.
- The most common example of a static member is main().
- main() is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.
- When objects of its class are declared, no copy of a static variable is made.
- Instead, all instances of the class share the same static variable.



Static members

```
class Example {  
    // Static variable shared among all instances  
    static int staticVariable = 0;  
  
    // Static method  
    static void staticMethod() {  
        System.out.println("This is a static method.");  
    }  
}
```



Static members

```
public class Main {  
    public static void main(String[] args) {  
        // Accessing the static variable without creating an instance  
        System.out.println("Static Variable: " + Example.staticVariable);  
  
        // Calling the static method without creating an instance  
        Example.staticMethod();  
  
        // Modifying the static variable  
        Example.staticVariable = 42;  
  
        // Accessing the modified static variable  
        System.out.println("Modified Static Variable: "+ Example.staticVariable);  
    }  
}
```



Final

- A field can be declared as **final**.
- Doing so prevents its contents from being modified, making it, essentially, a constant.
- This means that you must initialize a final field when it is declared.
- You can do this in one of two ways:
 - First, you can give it a value when it is declared.
 - Second, you can assign it a value within a constructor.
- The first approach is probably the most common.



Final

```
public class Example {  
    // First approach: Initializing final field when declared  
    final int constantValue = 42;  
  
    // Second approach: Initializing final field within a constructor  
    final int anotherConstantValue;  
  
    public Example(int value) {  
        this.anotherConstantValue = value;  
    }  
}
```



After methods are executed, all objects called within those methods become unreachable, which makes them eligible for garbage collection.

Variables scope

- A **scope** determines what objects are visible to other parts of your program.
- It also determines the lifetime of those objects.
- A **block** defines a scope, it is begun with an opening curly brace and ended by a closing curly brace.
- Thus, each time you start a new block, you are creating a new scope.
- It is not uncommon to think in terms of two general categories of scopes: global and local.
- However, these traditional scopes do not fit well with Java's strict, object-oriented model.



Variables scope

- In Java, the two major scopes are those **defined by a class** and those **defined by a method**.
- The scope defined by a method begins with its opening curly brace and ends with its closing curly brace.
- This block of code is called the method body.
- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- When we declare a variable within a scope, we are protecting that variable from unauthorized access and/or modification.
- The scope rules provide the foundation for encapsulation.
- A variable declared within a block is called a **local variable**.



Variables scope

- Scopes can be nested. For example, each time you create a block of code, you are creating a new, **nested scope**.
- When this occurs, the outer scope encloses the inner scope.
- This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true.
- variables declared within a method will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.



Variables scope

```
public class ScopeExample {  
    public static void main(String[] args) {  
        // Method scope variable  
        int x = 10;  
        if(x > 1) {  
            // Block scope variable  
            int y = 20;  
            System.out.println("Inside Block: x = " + x);  
            System.out.println("Inside Block: y = " + y);  
        }  
        // Variable y is not accessible here (outside its scope)  
        // Uncommenting the line below will result in a compilation error  
        // System.out.println("Outside Block: y = " + y);  
    }  
}
```



Method chaining

- **Method chaining** is a programming technique that allows invoking multiple methods on a single object in a single line of code.
- It enhances code readability and conciseness by creating a fluent interface.
- Basic syntax involves calling methods consecutively on the same object.
- Each method returns the object itself, enabling the chaining of subsequent methods, Example:

```
MyClass obj = new MyClass();  
obj.method1().method2().method3();
```

Method chaining

```
public class StringBuilderExample {  
    public static void main(String[] args) {  
        // Without Method Chaining  
        StringBuilder sb1 = new StringBuilder();  
        sb1.append("Hello");  
        sb1.append(" ");  
        sb1.append("World");  
        String result1 = sb1.toString();  
        System.out.println("Without Method Chaining: " + result1);  
  
        // With Method Chaining  
        StringBuilder sb2 = new StringBuilder();  
        String result2 = sb2.append("Hello").append(" ").append("World").toString();  
        System.out.println("With Method Chaining: " + result2);  
    }  
}
```



Method overloading

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be **overloaded**, and the process is referred to as **method overloading**.
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.



Method overloading #1

```
public class MathOperations {  
    // Overloaded method with two integer parameters  
    public static int add(int a, int b) {  
        return a + b;  
    }  
    // Overloaded method with two double parameters  
    public static double add(double a, double b) {  
        return a + b;  
    }  
    public static void main(String[] args) {  
        int sumInt = add(5, 10);  
        System.out.println("Sum of integers: " + sumInt);  
        double sumDouble = add(3.5, 2.7);  
        System.out.println("Sum of doubles: " + sumDouble);  
    }  
}
```



Method overloading #2

```
public class MathOperations {  
    // Overloaded method with two int parameters  
    public static int add(int a, int b) {  
        return a + b;  
    }  
    // Overloaded method with three int parameters  
    public static int add(int a, int b, int c) {  
        return a + b + c;  
    }  
    public static void main(String[] args) {  
        int sumInt = add(5, 10);  
        System.out.println("Sum of two integers: " + sumInt);  
        int sumThreeInt = add(2, 4, 6);  
        System.out.println("Sum of three integers: " + sumThreeInt);  
    }  
}
```



Overloading constructors

- In addition to overloading normal methods, you can also overload constructor methods.
- In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.



Overloading constructors

```
public class Person {  
    private String name;  
    private int age;  
    // Default constructor  
    public Person() {  
        this.name = "Unknown";  
        this.age = 0;  
    }  
    // Constructor with name parameter  
    public Person(String name) {  
        this.name = name;  
        this.age = 0;  
    }  
  
    //=> continued
```

```
    // Constructor with name,age parameters  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // Getter methods  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```



Overloading constructors

```
public class Main {  
  
    public static void main(String[] args) {  
        // Creating objects using different constructors  
        Person person1 = new Person();  
        Person person2 = new Person("John");  
        Person person3 = new Person("Jane", 25);  
  
        // Displaying information  
        System.out.println("Person 1: " + person1.getName() + ", " + person1.getAge());  
        System.out.println("Person 2: " + person2.getName() + ", " + person2.getAge());  
        System.out.println("Person 3: " + person3.getName() + ", " + person3.getAge());  
    }  
}
```



Method overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass.
- Method overriding occurs only when the names and the type signatures of the two methods are identical.
- If they are not, then the two methods are simply overloaded.



Method overriding

```
class Animal {  
    // Method in the superclass  
    void makeSound() {  
        System.out.println("Generic animal sound");  
    }  
}  
  
class Dog extends Animal {  
    // Overriding method in the subclass  
    void makeSound() {  
        System.out.println("Woof! Woof!");  
    }  
}
```



Method overriding

```
public class Main {  
    public static void main(String[] args) {  
        Animal genericAnimal = new Animal();  
        Dog myDog = new Dog();  
  
        // Calling the makeSound method on objects of both classes  
        genericAnimal.makeSound(); // Output: Generic animal sound  
        myDog.makeSound();          // Output: Woof! Woof!  
    }  
}
```



Thank you

