

Access Control for Methods and Variables

- The Java language provides four levels of access control: public, private, protected, and a default level specified by using none of these access control modifiers.
- These modifiers determine which variables and methods of a class are visible to other classes
- *Encapsulation is the process that prevents class variables from being read or modified by other classes. The only way to use these variables is by calling methods of the class, if they are available.*

Default Access

- A variable or method declared without any access control modifier is available to any other class in the same package.
- Any variable declared without a modifier can be read or changed by any other class in the same package.
- Any method declared the same way can be called by any other class in the same package.
- No other classes can access these elements in any way.
- This level of access control doesn't control much access.

Private Access

- A private instance variable can be used by methods in its own class but not by objects of any other class.
- Private methods can be called by other methods in their own class but cannot be called by any others.
- This restriction also affects inheritance:
- Neither private variables nor private methods are inherited by subclasses.
- Private variables are useful in two circumstances:
 - When other classes have no reason to use that variable
 - When another class could abuse by changing the variable in an inappropriate way
- **A big advantage of privacy is that it gives you a way to change the implementation of a class without affecting the users of that class.**

Public Access

- The public modifier makes a method or variable completely available to all classes.
- The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.
- Because of class inheritance, all public methods and variables of a class are inherited by its subclasses

Protected Access

- limit a method and variable to use by the following two groups:
 - Subclasses of a class
 - Other classes in the same package
- Protected access differs from default access this way; protected variables are available to subclasses, even if they aren't in the same package.
- Useful if you want to make it easier for a subclass to implement itself.

Comparing Levels of Access Control

Visibility	public	protected	default	private
From the same class	yes	yes	yes	yes
From any class in the same package	yes	yes	yes	no
From any class outside the package	yes	no	no	no
From a subclass in the same package	yes	yes	yes	no
From a subclass outside the same package	yes	yes	no	no

Access Control and Inheritance

- As a general rule, you cannot override a method in Java and make the new method more restrictively controlled than the original. You can, however, make it more public.
- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so the rules don't apply.

Final Classes, Methods, and Variables

- The final modifier is used with classes, methods, and variables to indicate that they will not be changed.
- A final variable cannot change in value. *constant variables (or just constants)*
- A final class cannot be subclassed. All methods in a final class automatically are final themselves, so you don't have to use a modifier in their declarations
- A final method cannot be overridden by any subclasses.
 - The most common reason to declare a method final is to make the class run more efficiently.
 - Normally, when java interpreter runs a method, it checks the current class to find the method first, checks its superclass second, and onward up the class hierarchy until the method is found. If a method is `final`, the Java compiler can put the executable bytecode of the method directly into any program that calls the method.
- Private methods are final without being declared that way because they can't be overridden in a subclass under any circumstance.

Abstract Classes and Methods

- In a class hierarchy, the higher the class, the more abstract its definition.
- A class at the top of a hierarchy of other classes can define only the behavior and attributes common to all the classes.
- More specific behavior and attributes are going to fall somewhere lower down the hierarchy.
- Abstract classes are classes that don't ever need to be instantiated directly.
- Instead, such a class serves as a place to hold common behavior and attributes shared by their subclasses
- ```
public abstract class Palette {
 // ...
}
```

# Abstract Classes and Methods

- Abstract classes can contain anything a normal class can, including constructor methods, because their subclasses might need to inherit the methods.
- Abstract classes can contain abstract methods, which are method signatures with no implementation.
- These methods are implemented in subclasses of the abstract class.
- Abstract methods are declared with the abstract modifier.
- You cannot declare an abstract method in a class that isn't itself abstract.
- If an abstract class has nothing but abstract methods, you're better off using an interface.

# Interfaces

- Interfaces, like abstract classes and methods, provide templates of behavior that other classes are expected to implement.
- They also offer significant advantages in class and object design that complements Java's single inheritance approach to object-oriented programming.
- A Java interface is a collection of abstract behavior that can be adopted by any class without being inherited from a superclass.
- An interface contains nothing but abstract method definitions and constants—there are no instance variables or method implementations.

# Interfaces and Classes

- Both are declared in source files and compiled into .class files
- In most cases, an interface can be used anywhere you can use a class
- Interfaces complement and extend the power of classes, and the two can be treated almost the same, but an interface cannot be instantiated: “new” only can create an instance of a nonabstract class.

# Implementing and Using Interfaces

- You can do two things with interfaces: Use them in your own classes and define your own.
- To use an interface, include the *implements* keyword as part of your class definition:
- ```
public class AnimatedSign extends javax.swing.JApplet  
implements Runnable {  
    //...  
}
```
- In this example, `javax.swing.JApplet` is the superclass, but the `Runnable` interface extends the behavior that it implements
- Because interfaces provide nothing but abstract method definitions, you then have to implement those methods in your own classes using the same method signatures from the interface.

Implementing and Using Interfaces

- To implement an interface, you must offer all the methods in that interface—you can't pick and choose the methods you need.
- By implementing an interface, you're telling users of your class that you support the entire interface.
- After your class implements an interface, subclasses of your class inherit those new methods and can override or overload them.
- If your class inherits from a superclass that implements a given interface, you don't have to include the implements keyword in your own class definition.

Implementing Multiple Interfaces

- Unlike with the singly inherited class hierarchy, you can include as many interfaces as you need in your own classes.
- Your class will implement the combined behavior of all the included interfaces.
- To include multiple interfaces in a class, just separate their names with commas:
- `public class AnimatedSign extends javax.swing.JApplet`
- `implements Runnable, Observable {`
`// ...`
`}`

Implementing Multiple Interfaces

- What happens if two different interfaces both define the same method?
- If the methods in each of the interfaces have identical signatures, you implement one method in your class, and that definition satisfies both interfaces.
- If the methods have different argument lists, it is a simple case of method overloading; you implement both method signatures, and each definition satisfies its respective interface definition.
- If the methods have the same argument lists but differ in return type, you cannot create a method that satisfies both. In this case, trying to compile a class that implements both interfaces produces a compiler error message. Running across this problem suggests that your interfaces have some design flaws that you might need to reexamine.

Creating and Extending Interfaces

- To create a new interface, you declare it like this:
- ```
public interface Expandable {
 public static final int increment = 10;
 long capacity = 15000; // becomes public static and final
 public abstract void expand(); //explicitly public and abstract
 void contract(); // effectively public and abstract
}
```

# Methods Inside Interfaces

- Those methods are supposed to be abstract and apply to any kind of class, but how can you define arguments to those methods? You don't know what class will be using them!
- What class should that argument be? It should be any object that implements the Trackable interface rather than a particular class and its subclasses.
- The solution is to declare the argument as simply Trackable in the interface:
- ```
public interface Trackable {  
    public abstract Trackable beginTracking(Trackable self);  
}
```

- Then, in an actual implementation for this method in a class, you can take the generic Trackable argument and cast it to the appropriate object:
- ```
public class Monitor implements Trackable {
 public Trackable beginTracking(Trackable self) {
 Monitor mon = (Trackable) self;
 // ...
 }
}
```

# Extending Interfaces

- You can organize interfaces into a hierarchy.
- When one interface inherits from another interface, that “subinterface” acquires all the method definitions and constants that its “superinterface” declared.
- To extend an interface, you use the extends keyword just as you do in a class definition:
- ```
interface PreciselyTrackable extends Trackable {  
    // ...  
}
```
- Note that unlike classes, the interface hierarchy has no equivalent of the Object class— there is no root superinterface from which all interfaces descend. Interfaces can either exist entirely on their own or inherit from another interface.
- Note also that unlike the class hierarchy, the inheritance hierarchy can have multiple inheritance. For example, a single interface can extend as many classes as it needs to and the new interface contains a combination of all its parent’s methods and constants.

Inner Classes

- you can define a class inside a class as if it were a method or a variable.
- These types of classes are called *inner classes*
- Inner classes are invisible to all other classes, which means that you don't have to worry about name conflicts between it and other classes.
- Inner classes can have access to variables and methods within the scope of a toplevel class that they would not have as a separate class.
- In many cases, an inner class is a short class file that exists only for a limited purpose

```
1: public class SquareTool {
2:     public SquareTool(String input) {
3:         try {
4:             float in = Float.parseFloat(input);
5:             Square sq = new Square(in);
6:             float result = sq.value;
7:             System.out.println("The square of " + input + " is " + result);
8:         } catch (NumberFormatException nfe) {
9:             System.out.println(input + " is not a valid number.");
10:        }
11:    }
12:
13:    class Square {
14:        float value;
15:
16:        Square(float x) {
17:            value = x * x;
18:        }
19:    }
20:
21:    public static void main(String[] arguments) {
22:        if (arguments.length < 1) {
23:            System.out.println("Usage: java SquareTool number");
24:        } else {
25:            SquareTool dr = new SquareTool(arguments[0]);
26:        }
27:    }
28: }
```

- java SquareTool 13
- The square of 13 is 169.0
- The Square class is given the name SquareTool\$Square.class by the Java compiler

Exercise

- True/False?
- private abstract methods and final abstract methods are legal in Java
- False : they're compile-time error messages. abstract methods must be overridden, and abstract classes must be subclassed, but neither of those two operations would be legal if they were also private or final.

Exercise

- If you create a subclass and override a public method, what access modifiers can you use with that method?
 - a. public only
 - b. public or protected
 - c. public, protected, or default access
- a. All public methods must remain public in subclasses.