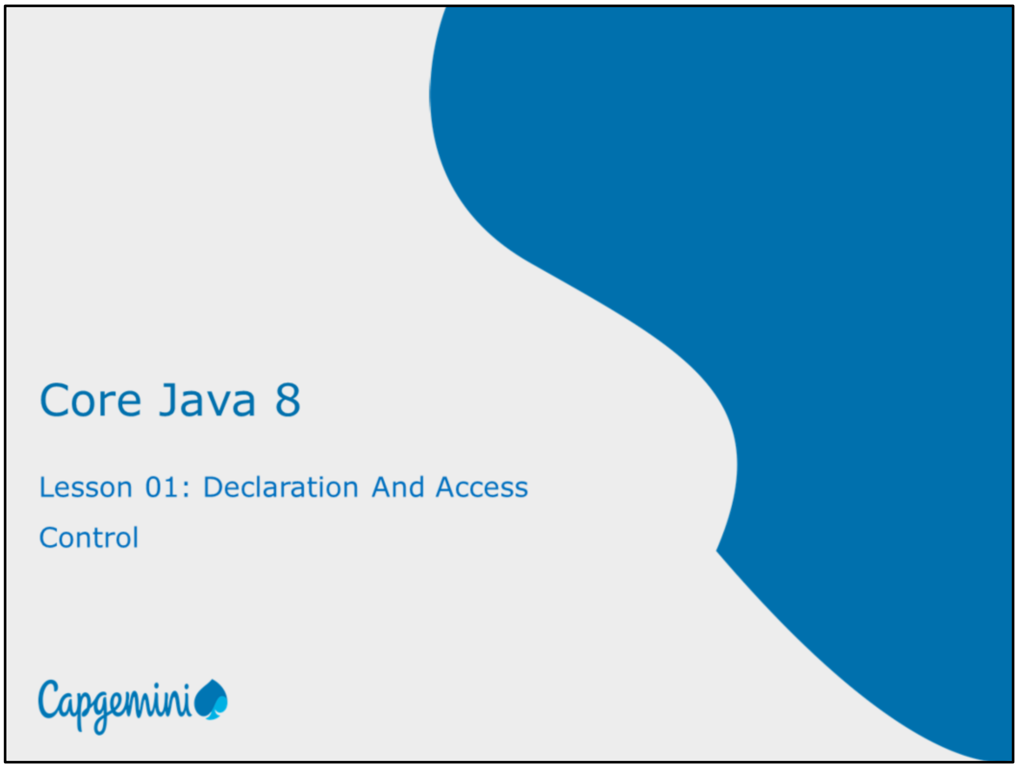
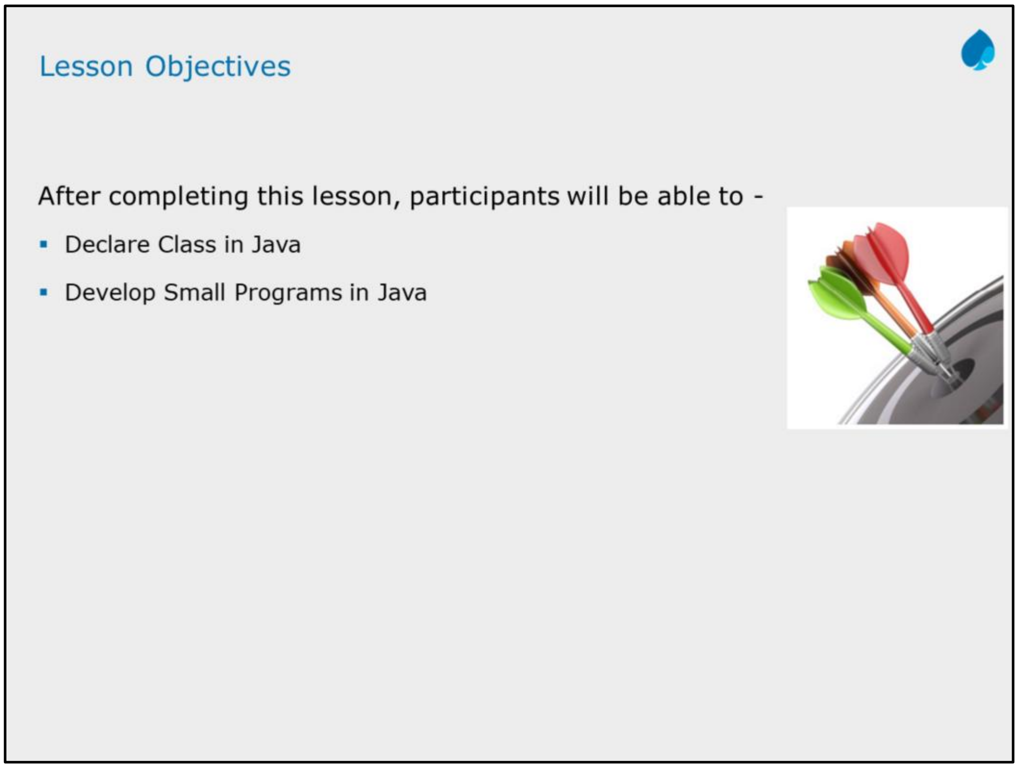


Core Java 8

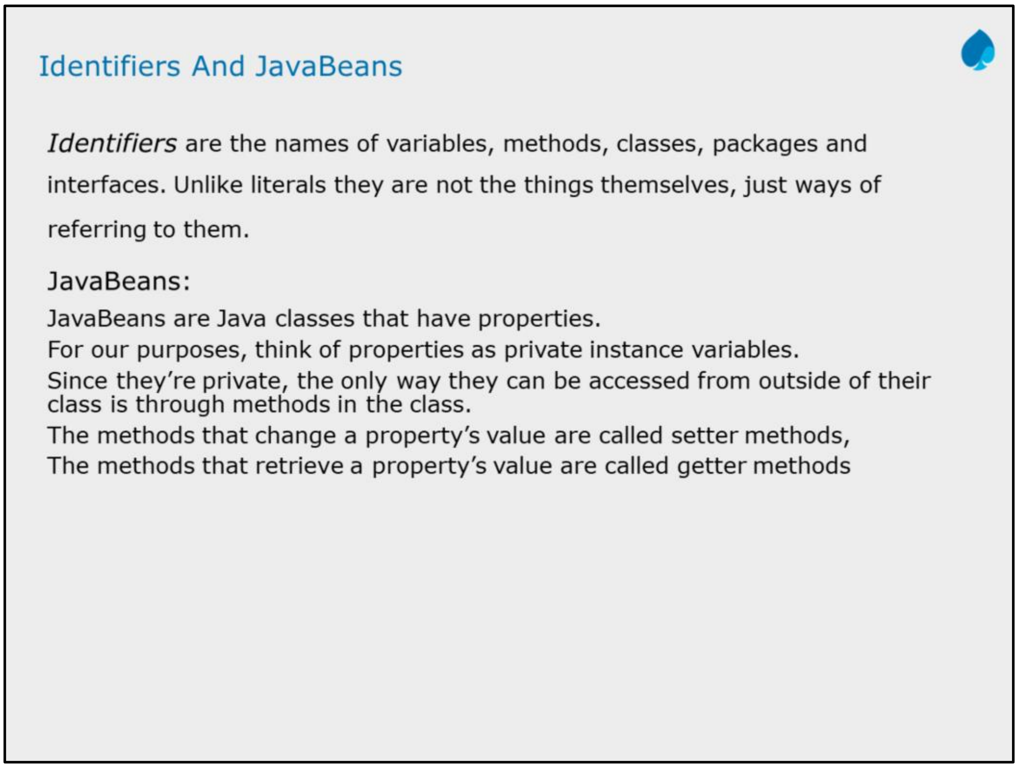
Lesson 01: Declaration And Access Control



Lesson Objectives

After completing this lesson, participants will be able to -

- Declare Class in Java
- Develop Small Programs in Java

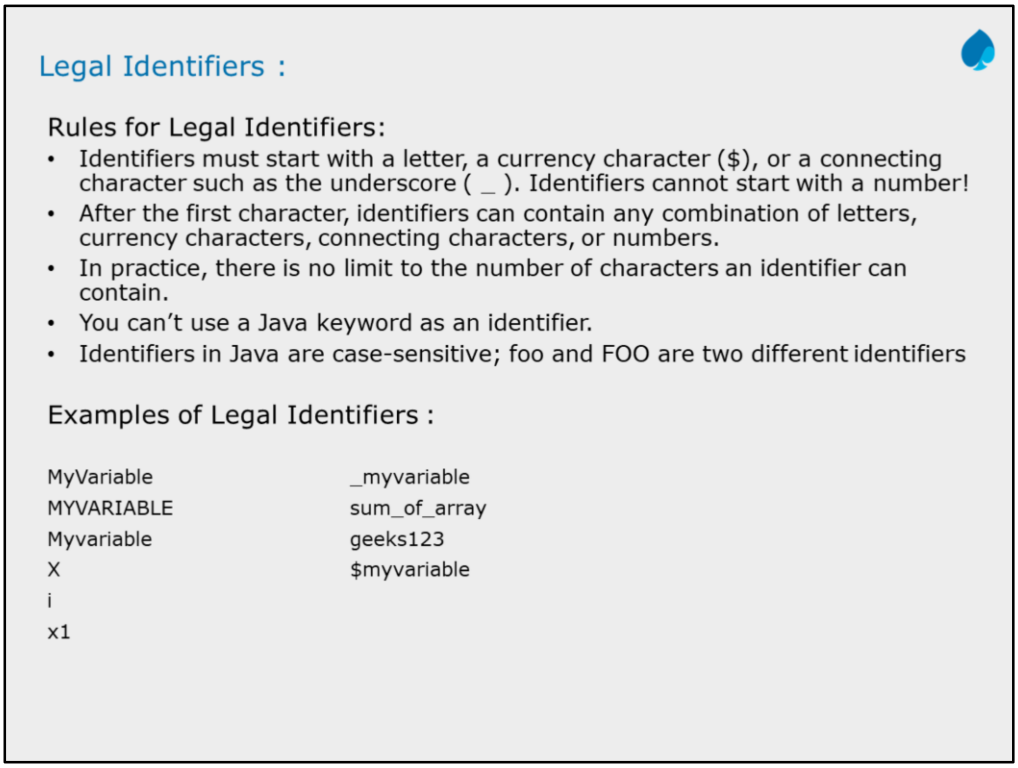


Identifiers And JavaBeans

Identifiers are the names of variables, methods, classes, packages and interfaces. Unlike literals they are not the things themselves, just ways of referring to them.

JavaBeans:

JavaBeans are Java classes that have properties.
For our purposes, think of properties as private instance variables.
Since they're private, the only way they can be accessed from outside of their class is through methods in the class.
The methods that change a property's value are called setter methods,
The methods that retrieve a property's value are called getter methods



Legal Identifiers :

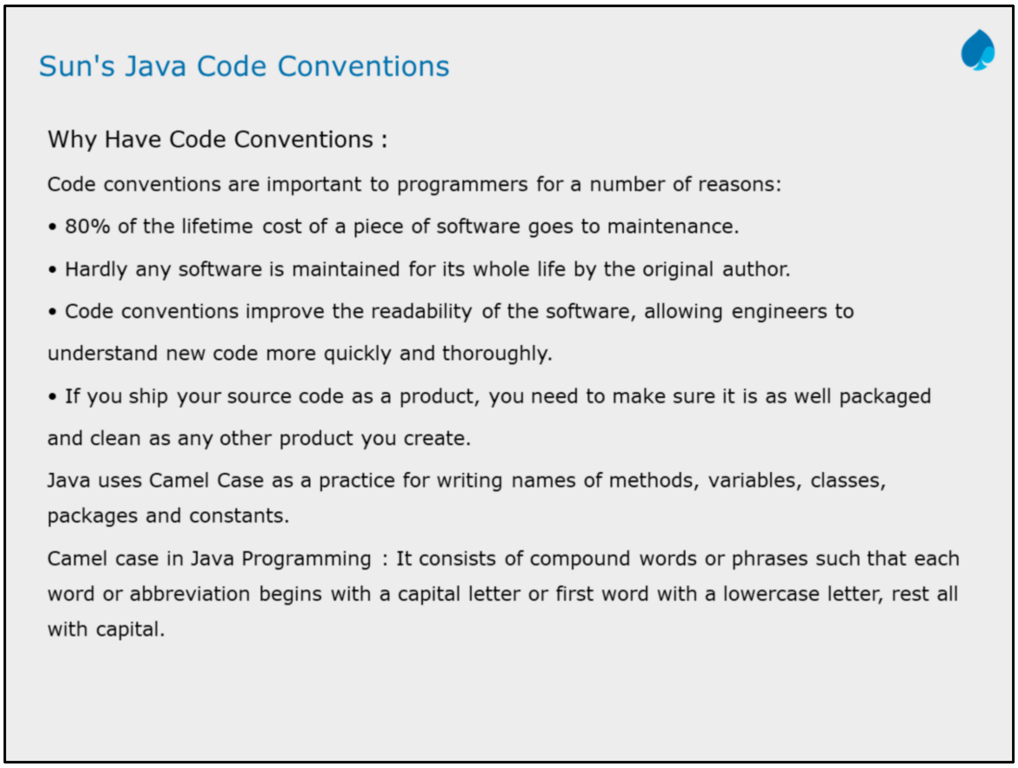
Rules for Legal Identifiers:

- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number!
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier.
- Identifiers in Java are case-sensitive; foo and FOO are two different identifiers

Examples of Legal Identifiers :

MyVariable
MYVARIABLE
Myvariable
X
i
x1

_myvariable
sum_of_array
geeks123
$myvariable



Sun's Java Code Conventions

Why Have Code Conventions :

Code conventions are important to programmers for a number of reasons:

• 80% of the lifetime cost of a piece of software goes to maintenance.

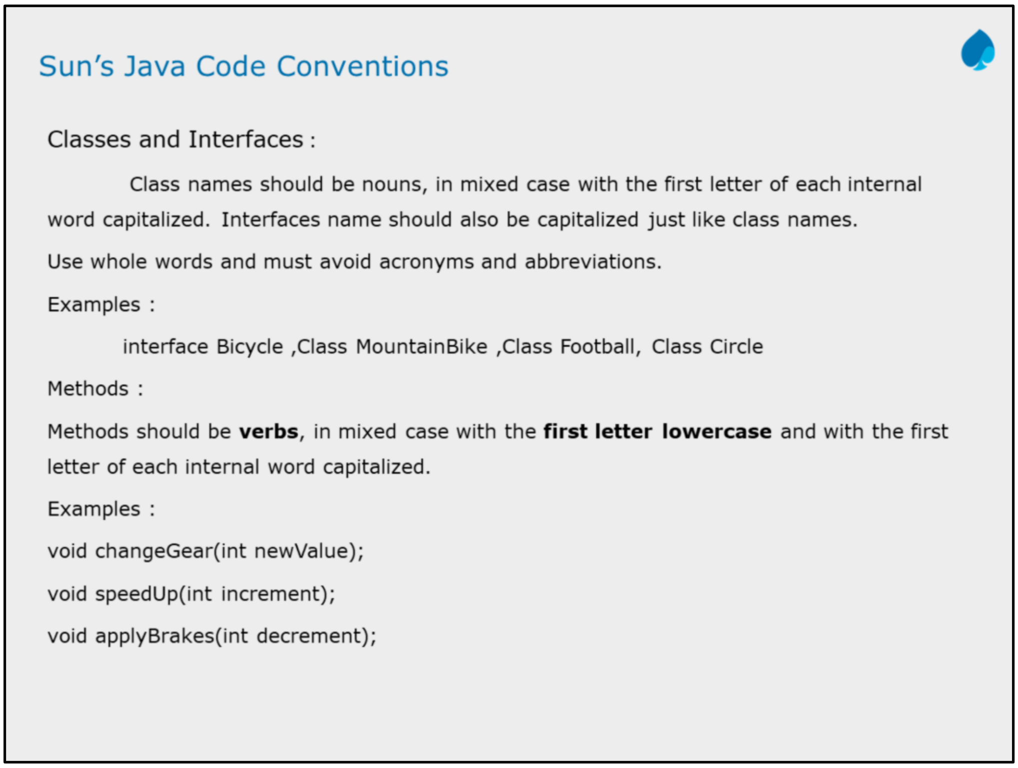
• Hardly any software is maintained for its whole life by the original author.

• Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

• If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Java uses Camel Case as a practice for writing names of methods, variables, classes, packages and constants.

Camel case in Java Programming : It consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital.



Sun’s Java Code Conventions

Classes and Interfaces :

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Interfaces name should also be capitalized just like class names.

Use whole words and must avoid acronyms and abbreviations.

Examples :

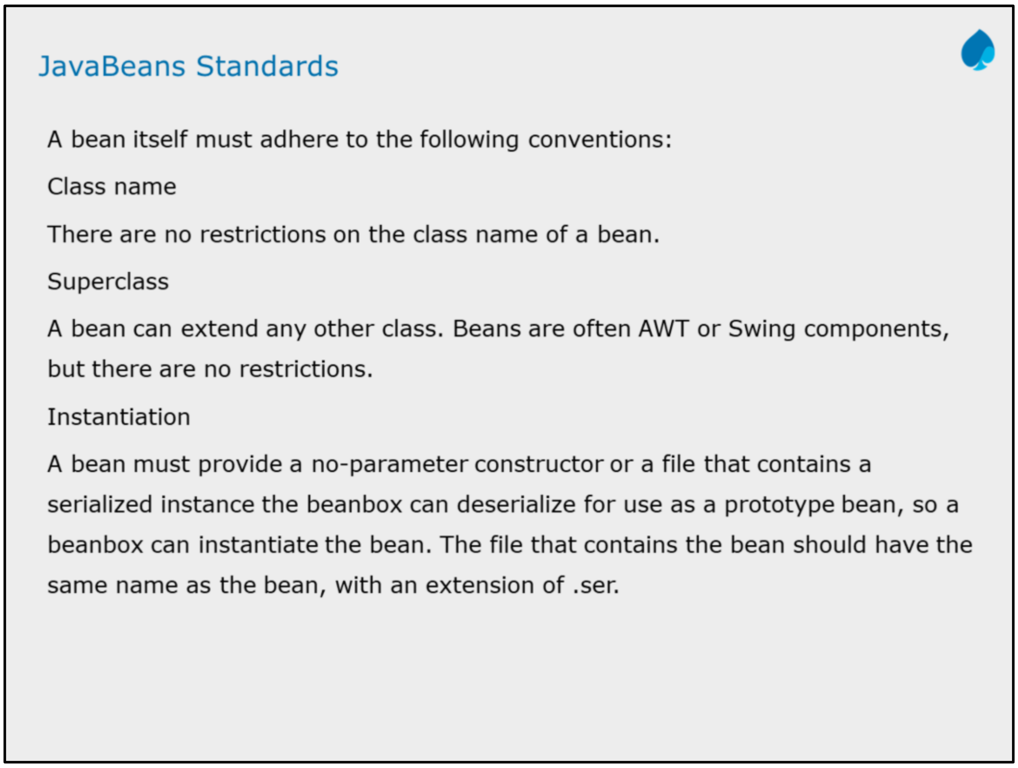
interface Bicycle ,Class MountainBike ,Class Football, Class Circle

Methods :

Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalized.

Examples :

```
void changeGear(int newValue);
void speedUp(int increment);
void applyBrakes(int decrement);
```



JavaBeans Standards

A bean itself must adhere to the following conventions:

Class name

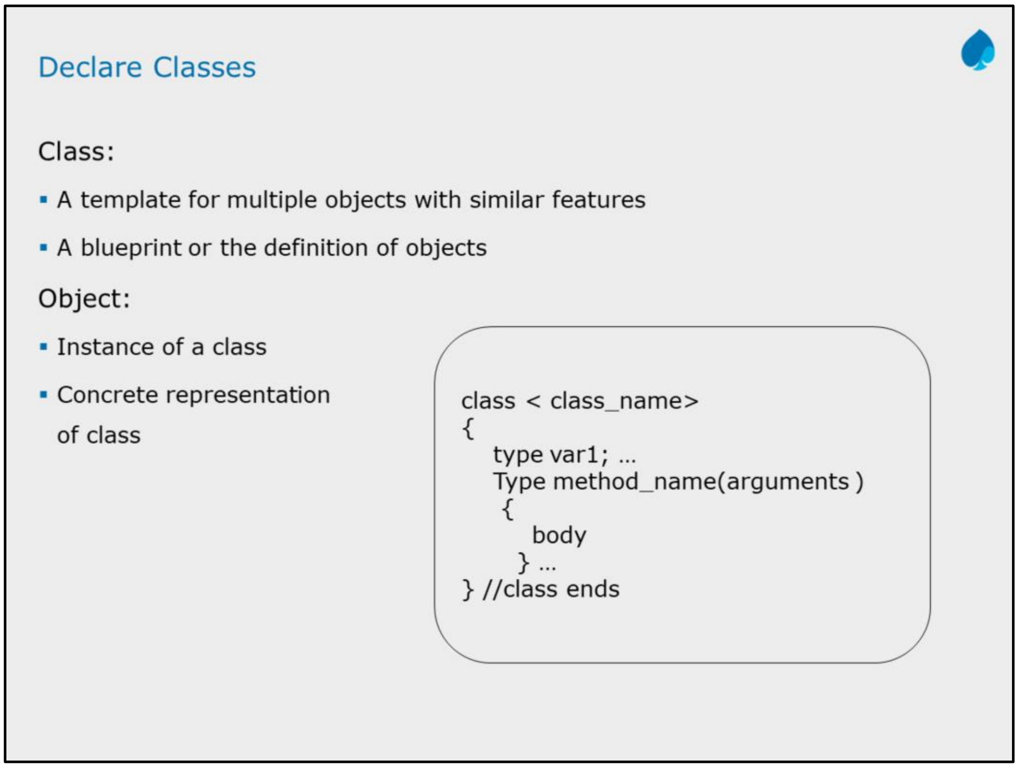
There are no restrictions on the class name of a bean.

Superclass

A bean can extend any other class. Beans are often AWT or Swing components, but there are no restrictions.

Instantiation

A bean must provide a no-parameter constructor or a file that contains a serialized instance the beanbox can deserialize for use as a prototype bean, so a beanbox can instantiate the bean. The file that contains the bean should have the same name as the bean, with an extension of .ser.



Declare Classes

Class:

- A template for multiple objects with similar features
- A blueprint or the definition of objects

Object:

- Instance of a class
- Concrete representation of class

```
class < class_name>
{
   type var1; ...
   Type method_name(arguments )
    {
       body
     } ...
} //class ends
```

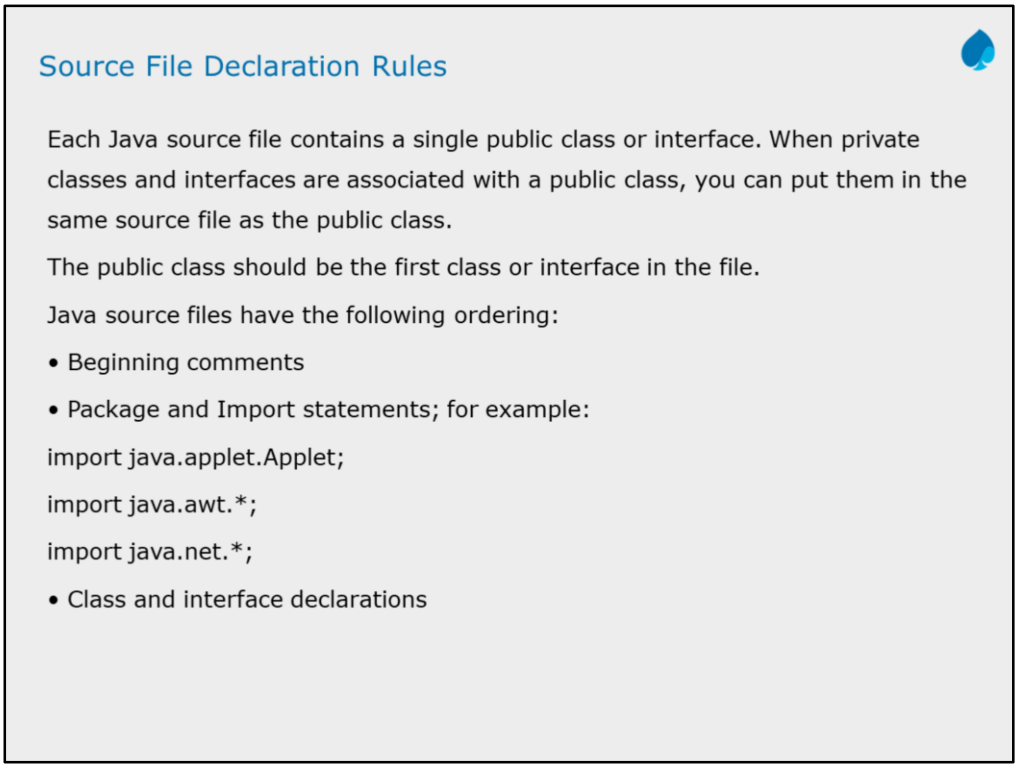
Classes describe objects that share characteristics, methods, relationships, and semantics. Each class has a name, attributes (its values determine state of an object), and operations (which provides the behavior for the object).

What is the relationship between objects and classes? What exists in real world is objects. When we classify these objects on the basis of commonality of structure and behavior, the result that we get are the classes.

Classes are “logical”. They don’t really exist in real world. When writing software programs, it is the classes that get defined first. These classes serve as a blueprint from which objects are created.

Reference: Refer to OOP material for a detailed discussion on Object-Oriented Programming Concept.

Note: Java does not have functions defined outside classes (as C++ does).



Source File Declaration Rules

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class.

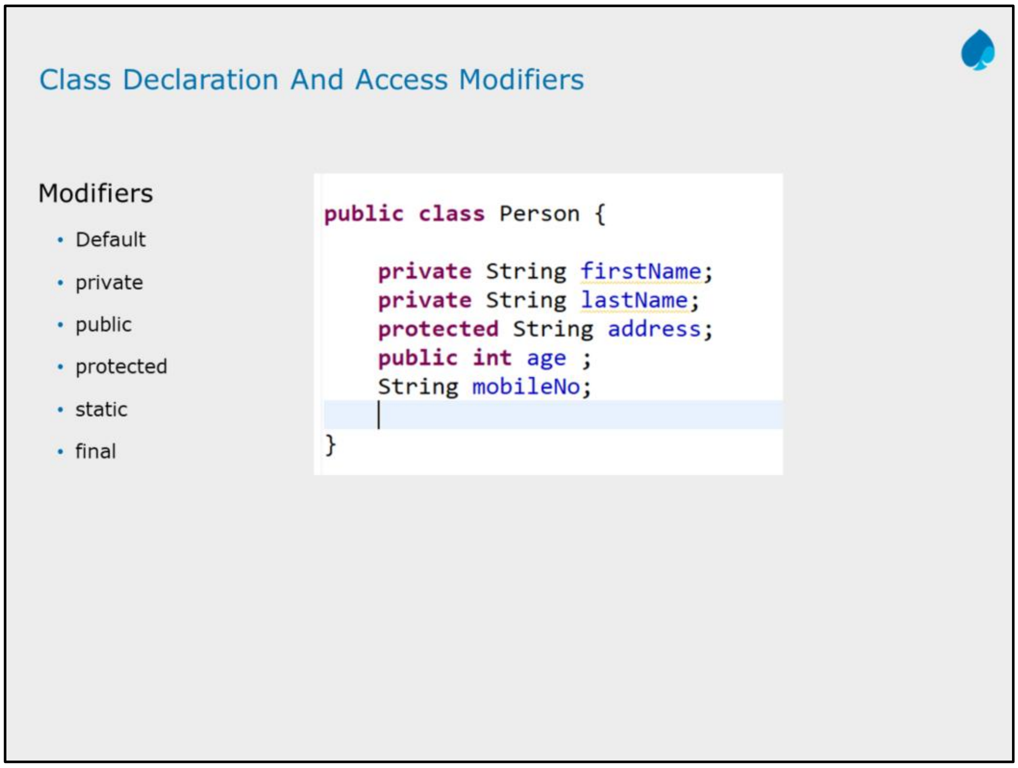
The public class should be the first class or interface in the file.

Java source files have the following ordering:

- Beginning comments
- Package and Import statements; for example:

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;
```

- Class and interface declarations



Class Declaration And Access Modifiers

Modifiers

- Default
- private
- public
- protected
- static
- final

```
public class Person {

    private String firstName;
    private String lastName;
    protected String address;
    public int age ;
    String mobileNo;

}
```

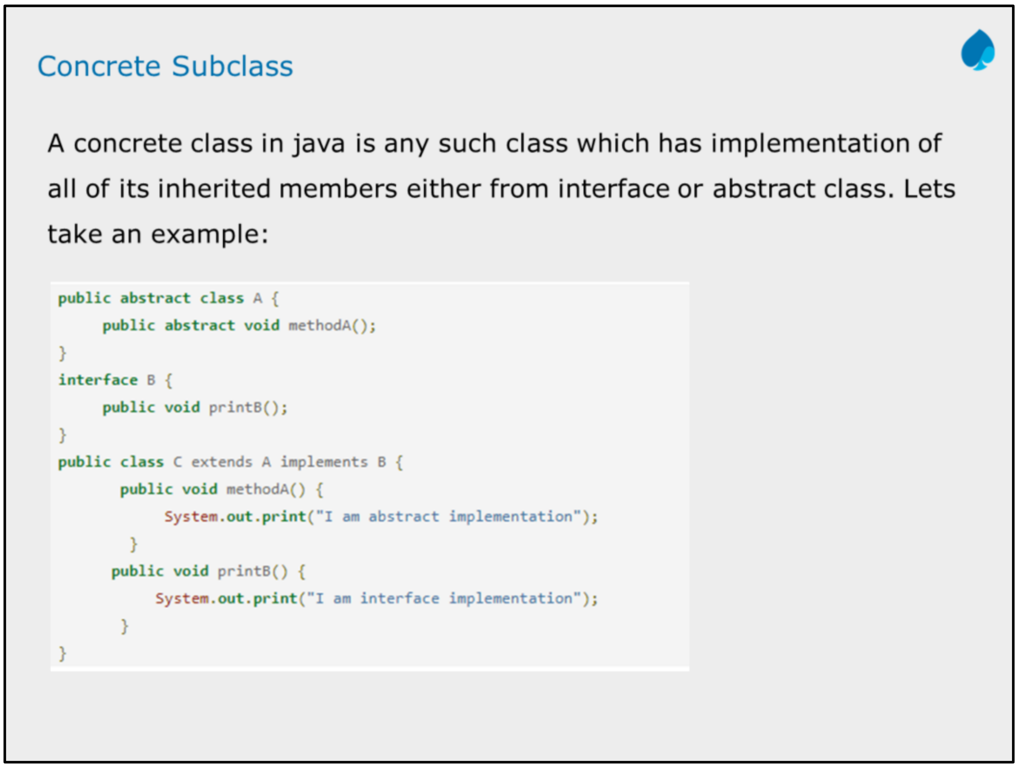
public access modifier
Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier
Fields, methods or constructors declared private (most restrictive) cannot be accessed outside an enclosing class. This modifier cannot be used for classes. It also cannot be used for fields and methods within an interface. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier
Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors, even if they are not a subclass of the protected member’s class. This modifier cannot be used for classes. It also cannot be used for fields and methods within an interface.

default access modifier
Default specifier is used when “no access modifier is present”. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

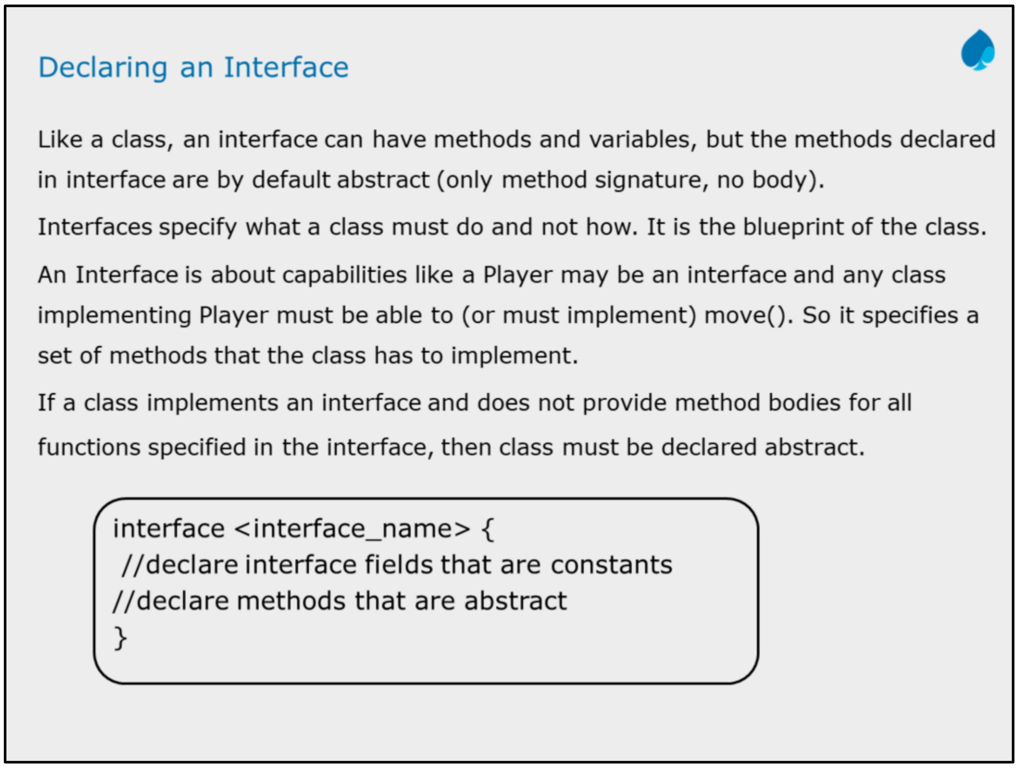
The table shown above is applicable only to members of classes. A class has only two possible access levels: default and public.
When a class is declared as public, it is accessible by any other code.
If a class has default access, then it can only be accessed by other code within its same package



Concrete Subclass

A concrete class in java is any such class which has implementation of all of its inherited members either from interface or abstract class. Lets take an example:

```
public abstract class A {
     public abstract void methodA();
}
interface B {
     public void printB();
}
public class C extends A implements B {
       public void methodA() {
            System.out.print("I am abstract implementation");
        }
      public void printB() {
           System.out.print("I am interface implementation");
       }
}
```



Declaring an Interface

Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).

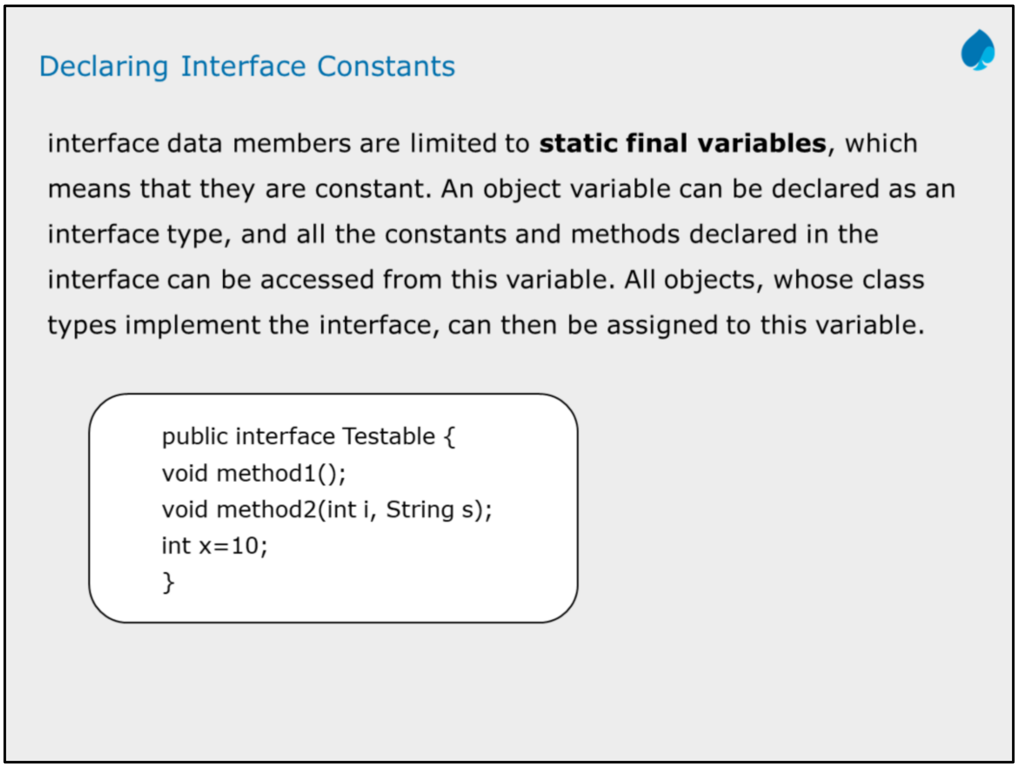
Interfaces specify what a class must do and not how. It is the blueprint of the class.

An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.

If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

```
interface <interface_name> {
 //declare interface fields that are constants
//declare methods that are abstract
}
```

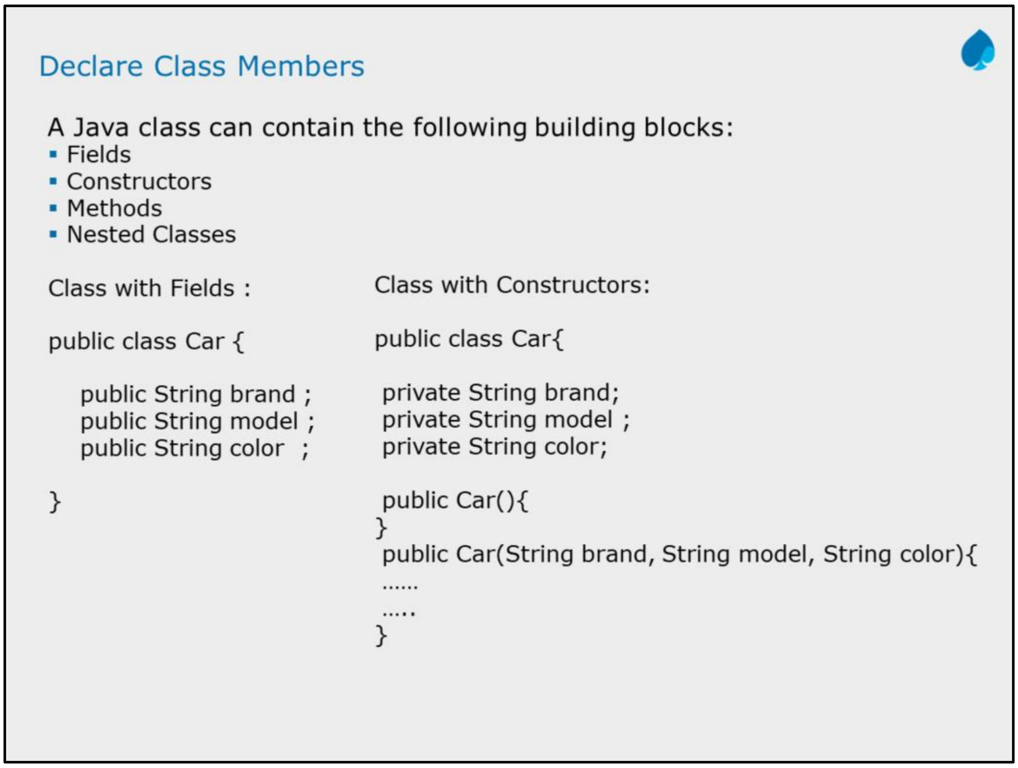
Interface:
You may come across a situation, in which you want to have different implementations of a method in different classes and delay the decision on which implementation of a method to execute until runtime. In java, the class where the method is defined must be present at compile time so that the compiler can check the signature of the method to ensure that the method call is legitimate. All the classes that could possibly be called for the aforementioned method need to share a common super class, so that the method can be defined in the super class and overridden by the individual subclasses. If you want to force every subclass to have its own implementation of the method, the method can be defined as an abstract one. Chances are you will want to move the method definition higher and higher up the inheritance hierarchy, so that more and more classes can override the same method.
Because of single inheritance, any Java class has only a single super class. It inherits variables and methods from all superclasses above it in the hierarchy. This makes sub-classing easier to implement and design, but it also can be restricting when you have similar behavior that must be duplicated across different branches of class hierarchy. Java solves the problem of shared behavior by using interfaces. An interface is a collection of method signatures (without implementations) and constant values. Interfaces are used to define a protocol behavior that can be implemented by any class hierarchy. Interfaces are abstract classes that are left completely unimplemented**.** That means, no methods in the class has been implemented. Using an interface, you can specify what a class must do, but not how it does.



Declaring Interface Constants

interface data members are limited to **static final variables**, which means that they are constant. An object variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed from this variable. All objects, whose class types implement the interface, can then be assigned to this variable.

```
public interface Testable {
void method1();
void method2(int i, String s);
int x=10;
}
```



Declare Class Members

A Java class can contain the following building blocks:

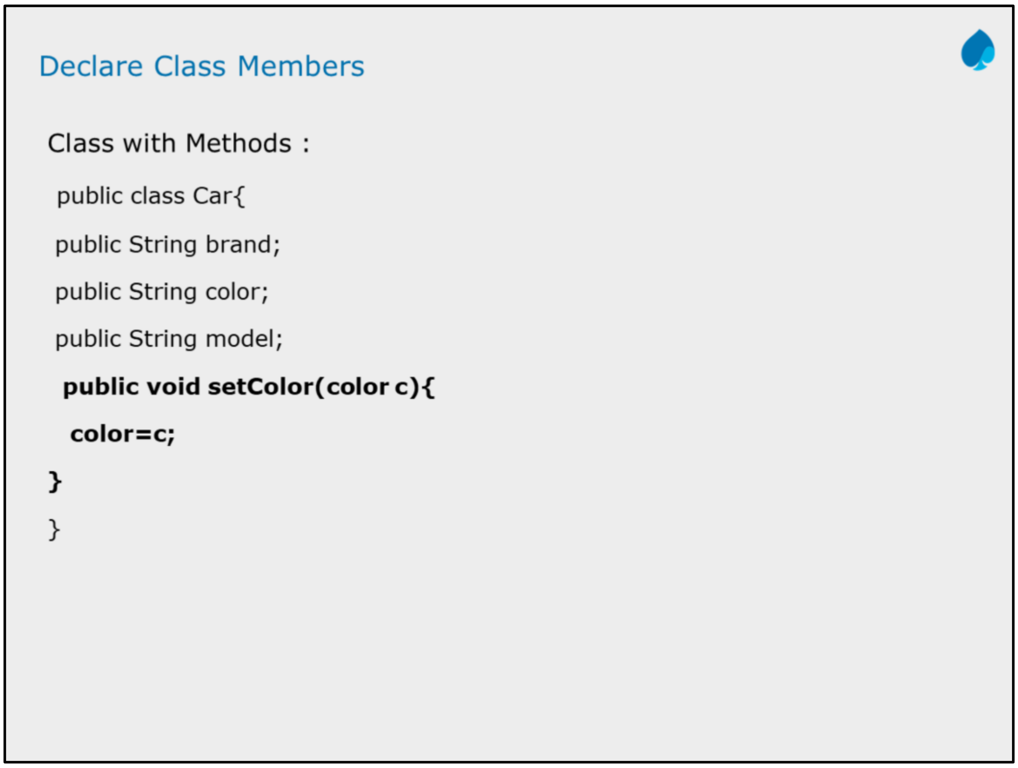
- Fields
- Constructors
- Methods
- Nested Classes

Class with Fields :

```
public class Car {

    public String brand ;
    public String model ;
    public String color  ;

}
```

Class with Constructors:

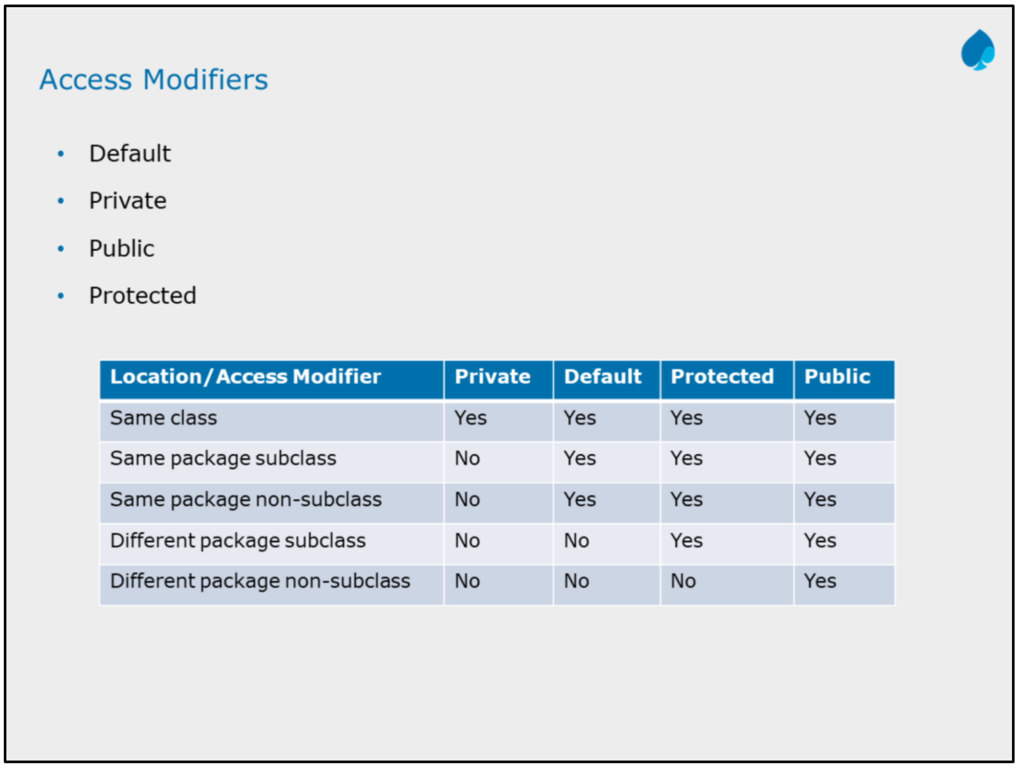
```
public class Car{

 private String brand;
 private String model ;
 private String color;

 public Car(){
}
 public Car(String brand, String model, String color){
 ……
 …..
}
```



Declare Class Members

Class with Methods :

```
public class Car{
public String brand;
public String color;
public String model;
 public void setColor(color c){
  color=c;
}
}
```



Access Modifiers

- Default
- Private
- Public
- Protected

Location/Access Modifier	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

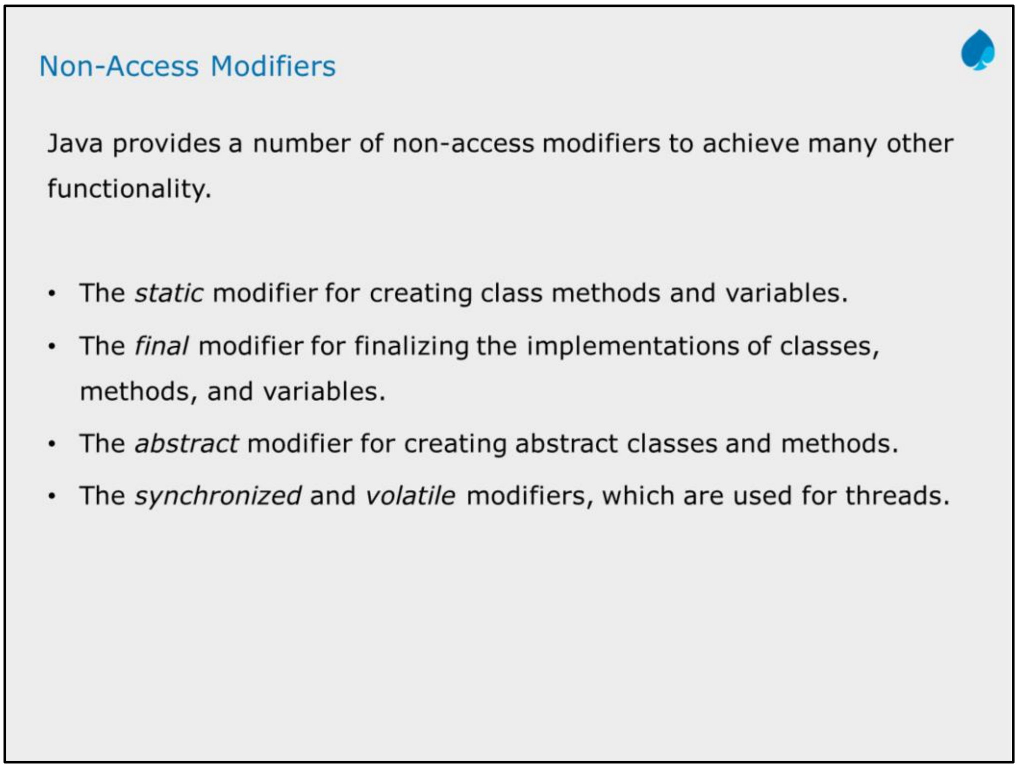
public access modifier
Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier
Fields, methods or constructors declared private (most restrictive) cannot be accessed outside an enclosing class. This modifier cannot be used for classes. It also cannot be used for fields and methods within an interface. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier
Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors, even if they are not a subclass of the protected member’s class. This modifier cannot be used for classes. It also cannot be used for fields and methods within an interface.

default access modifier
Default specifier is used when “no access modifier is present”. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

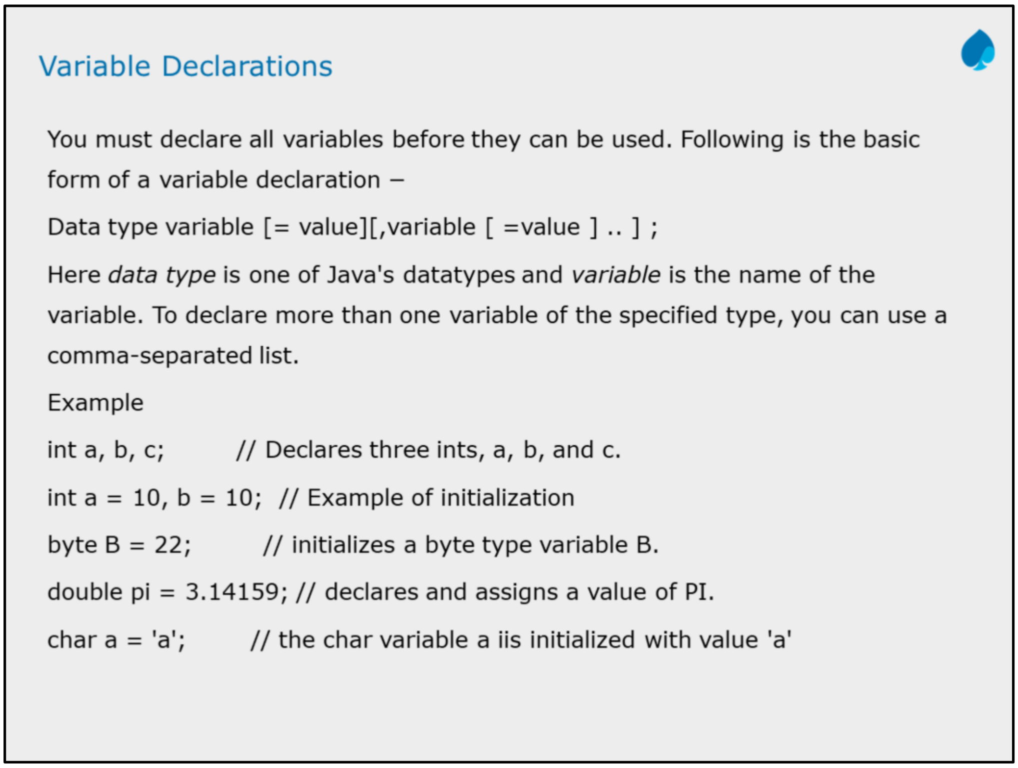
The table shown above is applicable only to members of classes. A class has only two possible access levels: default and public.
When a class is declared as public, it is accessible by any other code.
If a class has default access, then it can only be accessed by other code within its same package



Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.



Variable Declarations

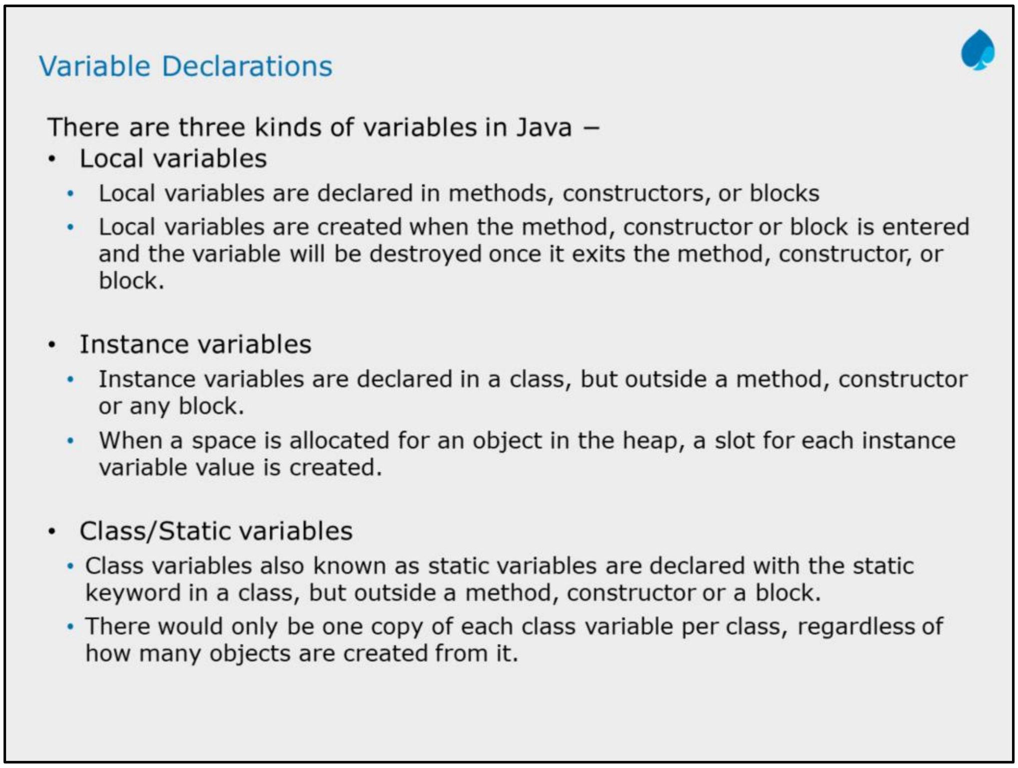
You must declare all variables before they can be used. Following is the basic form of a variable declaration –

Data type variable [= value][,variable [=value] ..] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Example

```
int a, b, c;          // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;          // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';         // the char variable a iis initialized with value 'a'
```



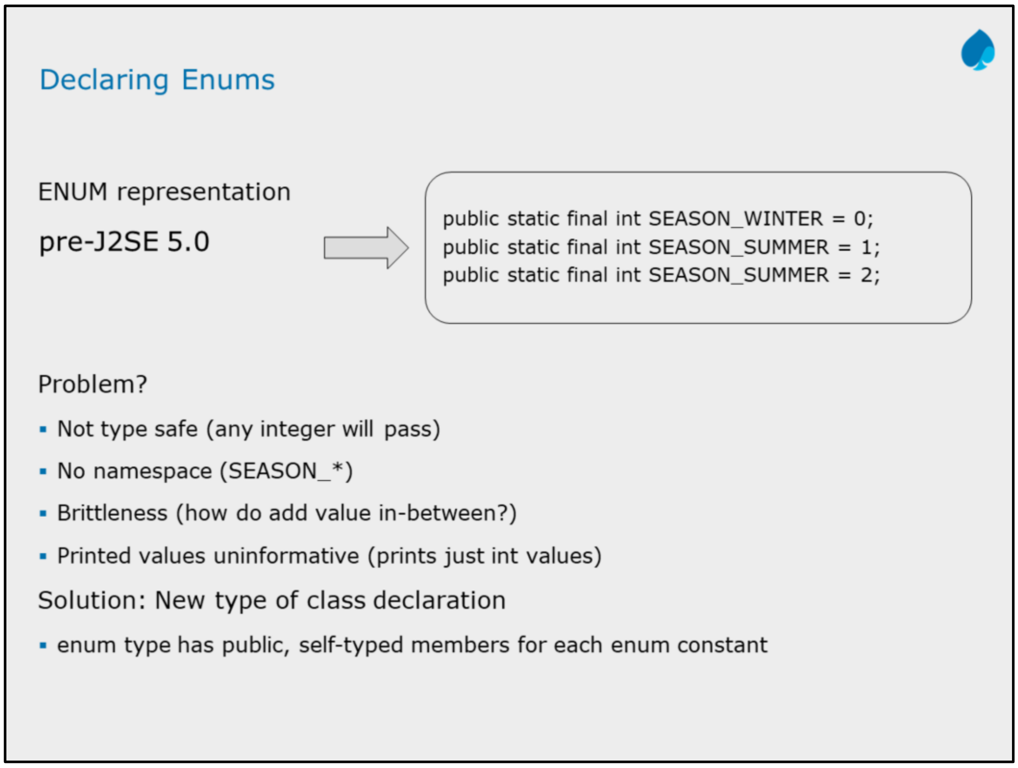
Variable Declarations

There are three kinds of variables in Java –

- Local variables
  - Local variables are declared in methods, constructors, or blocks
  - Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.

- Instance variables
  - Instance variables are declared in a class, but outside a method, constructor or any block.
  - When a space is allocated for an object in the heap, a slot for each instance variable value is created.

- Class/Static variables
  - Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
  - There would only be one copy of each class variable per class, regardless of how many objects are created from it.



Declaring Enums

ENUM representation

pre-J2SE 5.0

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SUMMER = 1;
public static final int SEASON_SUMMER = 2;
```

Problem?

- Not type safe (any integer will pass)
- No namespace (SEASON_*)
- Brittleness (how do add value in-between?)
- Printed values uninformative (prints just int values)

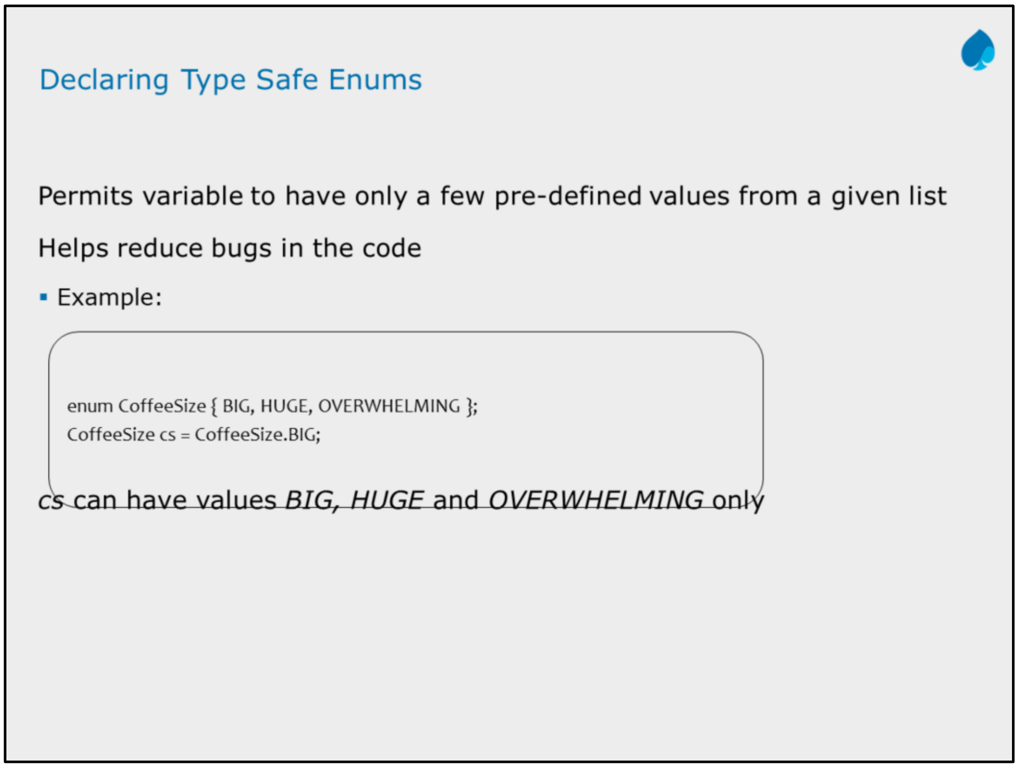
Solution: New type of class declaration

- enum type has public, self-typed members for each enum constant

In pre Java 5, the standard way to represent an enumerated type was the int Enum pattern. An example shown in the box above.

But this pattern suffers from problems such as:
Not typesafe – A season is just an int you can pass in any other int value where a season is required, or add two seasons together (makes no sense)!
No namespace - Constants of an int enum must be prefixed with a string (eg SEASON_) to avoid collisions with other int enum types.
Brittleness - Since int enums are compile-time constants, they are compiled into clients that use them. If a new constant is added between two existing constants or the order is changed, clients must be recompiled.
Printed values are uninformative - Since they are just ints, printing one out will get you a number, which tells you nothing about what it represents, or what type it is.

In J2SE5 the enumerations have become separate classes. Thus, they are type-safe, flexible to use. For example the above enum is now represented as :
enum Season { WINTER, SPRING, SUMMER, FALL }



Declaring Type Safe Enums

Permits variable to have only a few pre-defined values from a given list

Helps reduce bugs in the code

- Example:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
CoffeeSize cs = CoffeeSize.BIG;
```

cs can have values *BIG, HUGE* and *OVERWHELMING* only

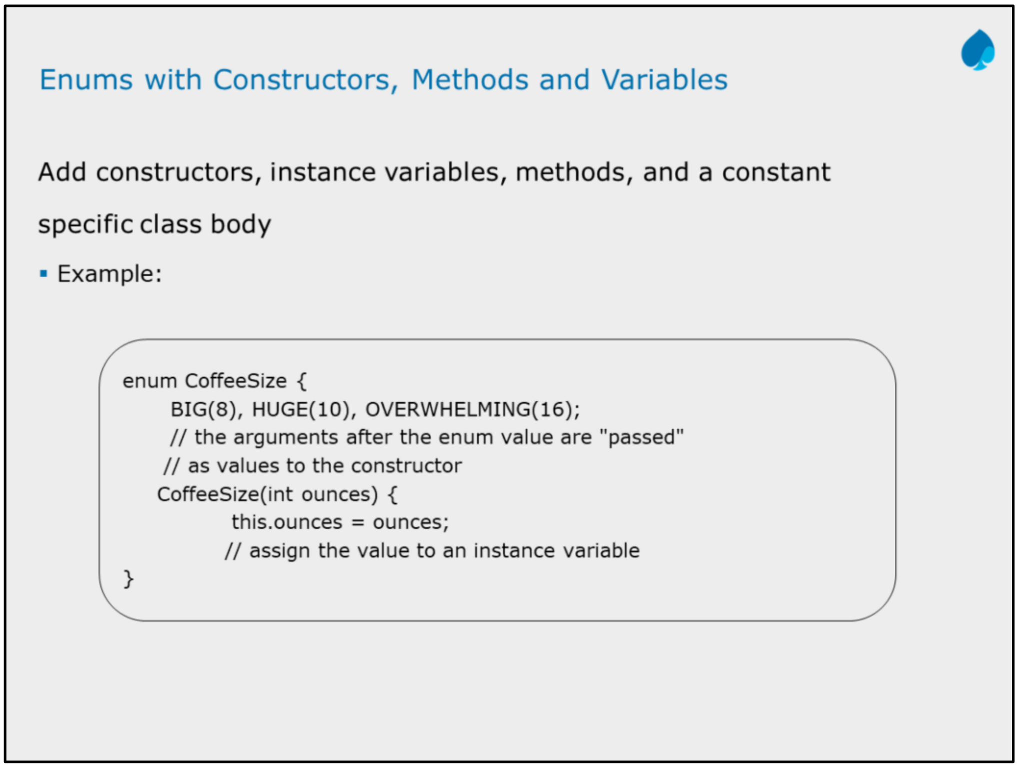
Enum lets you restrict a variable to having one of only a few pre-defined values—in other words, one value from an enumerated list. This can help reduce the bugs in your code. For instance, in your coffee shop application you might want to restrict your size selections to BIG, HUGE, and OVERWHELMING. If you let an order for a LARGE or a GRANDE slip in, it might cause an error. See the declaration above. With this, you can guarantee that the compiler will stop you from assigning anything to a CoffeeSize except BIG, HUGE, or OVERWHELMING.
Then, the only way to get a CoffeeSize is with a statement like the following:

```
CoffeeSize cs = CoffeeSize.BIG;
```

It is not required that enum constants be in all upper case, but borrowing from the Sun code convention that constants are named in upper case, it is a good idea.

Important point to ponder about enums:

enum can be declared with only a public or default modifier.
enums are not strings or integer type.
Enums can be declared as their own class, or enclosed in another class, and that the syntax for accessing an enum's members depends on where the enum was declared.
enum cannot be declared in functions.
A semicolon after an enum is optional



Enums with Constructors, Methods and Variables

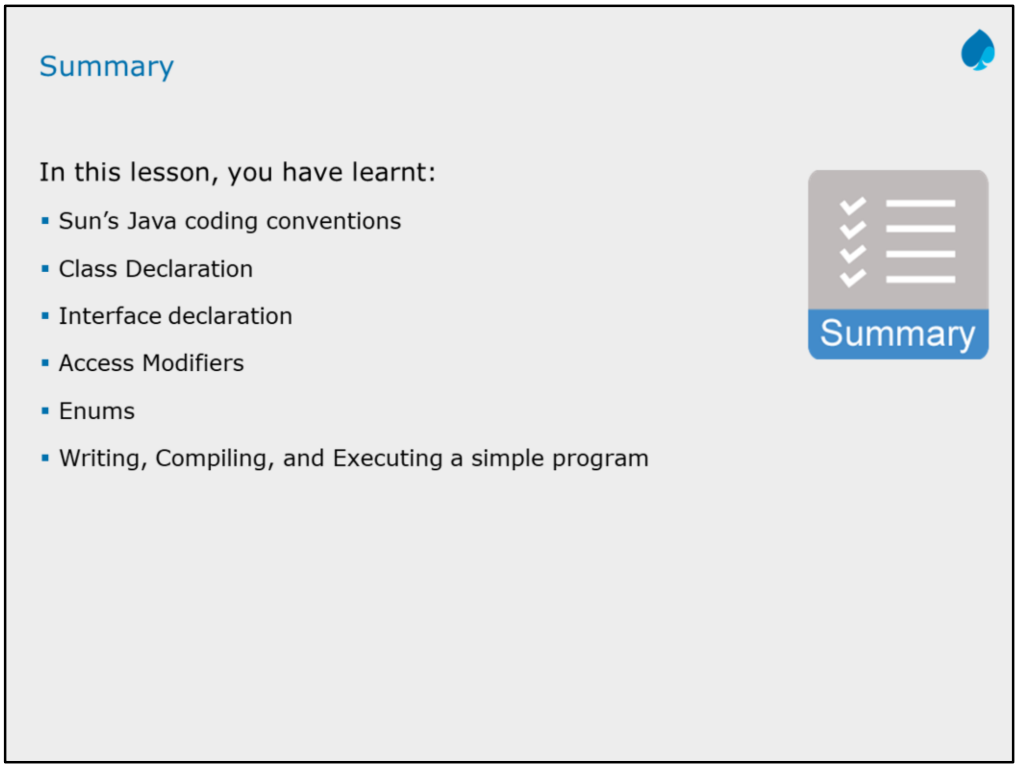
Add constructors, instance variables, methods, and a constant specific class body

- Example:

```
enum CoffeeSize {
       BIG(8), HUGE(10), OVERWHELMING(16);
       // the arguments after the enum value are "passed"
      // as values to the constructor
     CoffeeSize(int ounces) {
               this.ounces = ounces;
              // assign the value to an instance variable
}
```

Because an enum really is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods, and something really strange known as a constant specific class body. To understand why you might need more in your enum, think about this scenario: imagine you want to know the actual size, in ounces, that map to each of the three CoffeeSize constants.

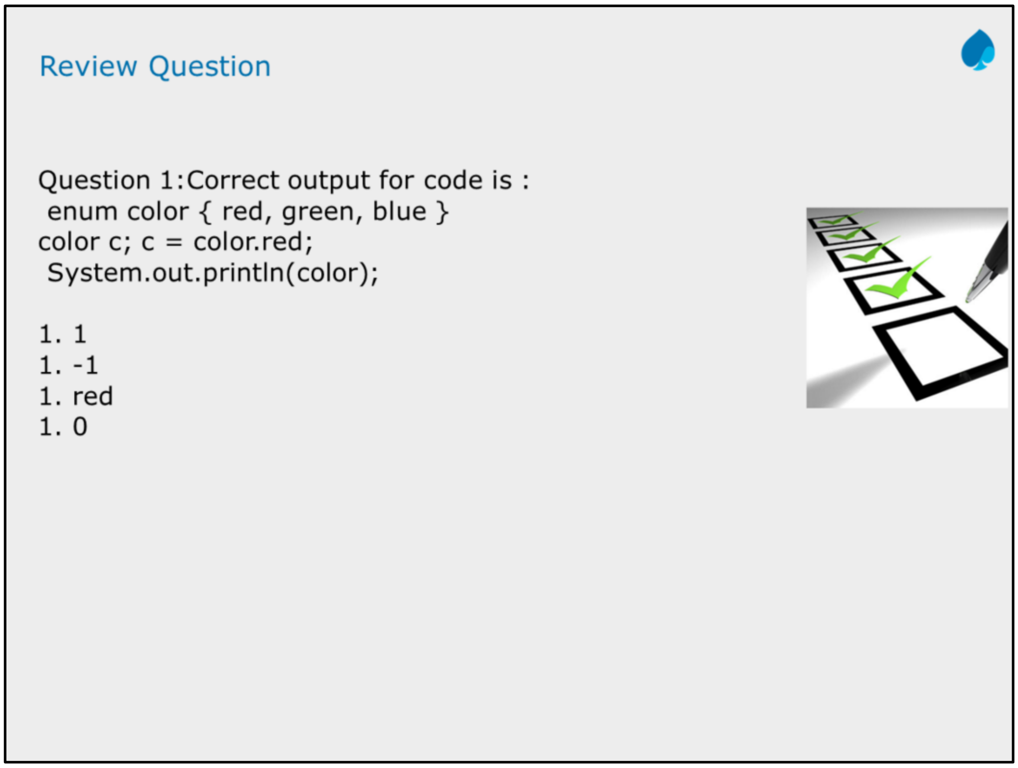
For example, you want to know that BIG is 8 ounces, HUGE is 10 ounces, and OVERWHELMING is a whopping 16 ounces. You could make some kind of a lookup table, using some other data structure, but that would be a poor design and hard to maintain. The simplest way is to treat your enum values (BIG, HUGE, and OVERWHELMING), as objects that can each have their own instance variables. Then you can assign those values at the time the enums are initialized, by passing a value to the enum constructor.



Summary

In this lesson, you have learnt:

- Sun's Java coding conventions
- Class Declaration
- Interface declaration
- Access Modifiers
- Enums
- Writing, Compiling, and Executing a simple program



Review Question

Question 1:Correct output for code is :

```
 enum color { red, green, blue }
color c; c = color.red;
 System.out.println(color);
```

1. 1
1. -1
1. red
1. 0