Class and Objects

Class Examples

```
public class Bicycle {
    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Declaring Classes

```
class MyClass {
      // field, constructor, and
      // method declarations
}
```

The *class body* contains all the code that provides for the life cycle of the objects created from the class:

- constructors for initializing new objects,
- declarations for the fields that provide the state of the class and its objects,
- methods to implement the behavior of the class and its objects.

Declaring Classes

```
class MyClass extends MySuperClass implements YourInterface {

    // field, constructor, and
    // method declarations
}
```

- MyClass is a subclass of MySuperClass
- It implements the YourInterface interface.

Declaring Classes

- Modifiers : *public*
- The class name, with the initial letter capitalized by convention.
- The name of the class's parent (superclass), if any, preceded by the keyword *extends*.
- A class can only *extend* (subclass) one parent.
- A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*.
- A class can *implement* more than one interface.
- The class body, surrounded by braces, {}.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class — these are called *fields*.
- Variables in a method or block of code — these are called *local variables*.
- Variables in method declarations — these are called *parameters*.

Declaring Member Variables

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

Access Modifiers

Public modifier—the field is accessible from all classes.

Private modifier—the field is accessible only within its own class.

- In the spirit of encapsulation, it is common to make fields private.
- This means that they can only be *directly* accessed from the Bicycle class.
- We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values.

Benefits of Using Objects

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
    public int getCadence() {
        return cadence;
    }
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public int getGear() {
        return gear;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public int getSpeed() {
        return speed;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Types

All variables must have a type:

- You can use primitive types such as int, float, boolean, etc.
- Or you can use reference types, such as strings, arrays, or objects.

Variable Names

- All variables follow Variable-Naming rules.
- The same naming rules and conventions are used for method and class names, except that
  - the first letter of a class name should be capitalized
  - the first (or only) word in a method name should be a verb.

Defining Methods

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are:

- the method ‘s return type
- name
- a pair of parentheses, ( )
- a body between braces, { }

Defining Methods

More generally, method declarations have six components, in order:

- Modifiers: public, private
- The return type
  - the data type of the value returned by the method
  - or void if the method does not return a value.
- The method name—the rules for field names apply to method names as well, but the convention is a little different.

Defining Methods

- The parameter list in parenthesis:
  - a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, ( ).
  - If there are no parameters, you must use empty parentheses.
- An exception list
- The method body, enclosed between braces—the method's code, including the declaration of local variables

Method Signature

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

Example: calculateAnswer(double, int, double, double)

Naming a Method

- By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc.
- In multi-word names, the first letter of each of the second and following words should be capitalized.
- Here are some examples:

run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty

Overloading Methods

- Typically, a method has a unique name within its class.
- However, a method might have the same name as other methods due to *method overloading*.
- Java can distinguish between methods with different parameter lists.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Overloading Methods

- Overloaded methods are differentiated by the number and the type of the arguments passed into the method.
- You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.
- The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.
- Overloaded methods should be used sparingly, as they can make code much less readable.

Constructors

- A class contains constructors that are invoked to create objects from the class blueprint.
- Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
}
```

Constructors

- To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

- new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.
- Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle( ) {
        gear = 1;
        cadence = 10;
        speed = 0;
}
```

Constructors

- You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart.
- You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors.
- You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Passing Information to a Method or a Constructor

```
public double computePayment(
                  double loanAmt,
                  double rate,
                  double futureValue,
                  int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
                    - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                    - ((futureValue * partial1) / denominator);
    return answer;
}
```

Parameters

- *Parameters* refers to the list of variables in a method declaration.
- *Arguments* are the actual values that are passed in when the method is invoked.
- When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

- primitive data types, such as doubles, floats, and integers
- reference data types, such as objects and arrays.

```
public Polygon polygonFrom(Point[] corners) {
    // method body goes here
}
```

- The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.

Arbitrary Number of Arguments

- You can use a construct called *varargs* to pass an arbitrary number of values to a method.
- You use *varargs* when you don't know how many of a particular type of argument will be passed to the method.
- It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).
- To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name.

Arbitrary Number of Arguments

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
                    * (corners[1].x - corners[0].x)
                    + (corners[1].y - corners[0].y)
                    * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);

    // more method body code follows that creates and returns a
    // polygon connecting the Points
}
```

- The method can then be called with any number of that parameter, including none.

Parameter Names

- When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.
- The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor.
- It cannot be the name of a local variable within the method or constructor.
- A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field.
- Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field.

Passing Primitive Data Type Arguments

- Primitive arguments, such as an int or a double, are passed into methods *by value*.
- This means that any changes to the values of the parameters exist only within the scope of the method.
- When the method returns, the parameters are gone and any changes to them are lost.

Passing Primitive Data Type Arguments

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        // invoke passMethod() with
        // x as argument
        passMethod(x);

        // print x to see if its
        // value has changed
        System.out.println("After invoking passMethod, x = " + x);

    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

When you run this program, the output is:

```
    After invoking passMethod, x = 3
```

Passing Primitive Data Type Arguments

- Reference data type parameters, such as objects, are also passed into methods *by value*.
- This means that when the method returns, the passed-in reference still references the same object as before.
- *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

Passing Reference Data Type Arguments

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

moveCircle(myCircle, 23, 56)

Inside the method, circle initially refers to myCircle.

The method changes the x and y coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively.

These changes will persist when the method returns.

Passing Reference Data Type Arguments

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Then circle is assigned a reference to a new Circle object with x = y = 0.

This reassignment has no permanence, however, because the reference was passed in by value and cannot change.

Within the method, the object pointed to by circle has changed,

but, when the method returns, myCircle still references the same Circle object as before the method was called.

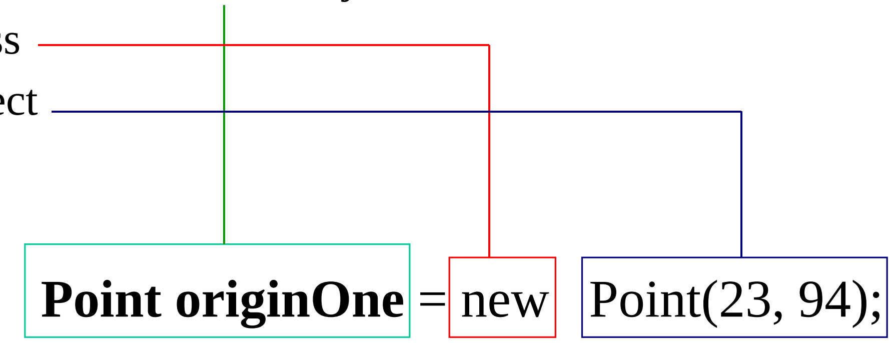
Objects

- A typical Java program creates many objects, which interact by invoking methods.
- Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declare and create a point object and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        // display rectOne's width, height, and area
        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        // set rectTwo's position
        rectTwo.origin = originOne;

        // display rectTwo's position
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

        // move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}
```

Creating Objects

- Declaring a Variable to Refer to an Object
- Instantiating a Class
- Initializing an Object



```
Point originOne = new Point(23, 94);
```

Declaring a Variable to Refer to an Object

If you declare a reference variable as follows:

```
Point originOne;
```

Its value will be undetermined until an object is actually created and assigned to it.

Simply declaring a reference variable does not create an object.

You must assign an object to originOne before you use it in your code. Otherwise, you will get a compiler error.

Instantiating a Class

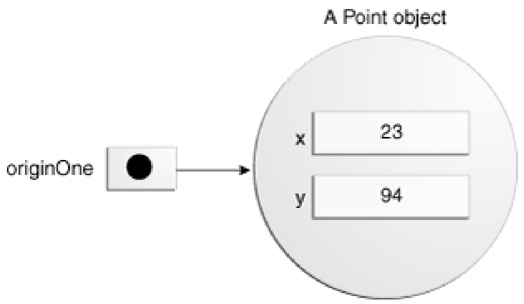
- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
- The new operator also invokes the object constructor.
- The new operator requires a single, postfix argument: a call to a constructor.
- The name of the constructor provides the name of the class to instantiate.
- The new operator returns a reference to the object it created.

Initializing an Object

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

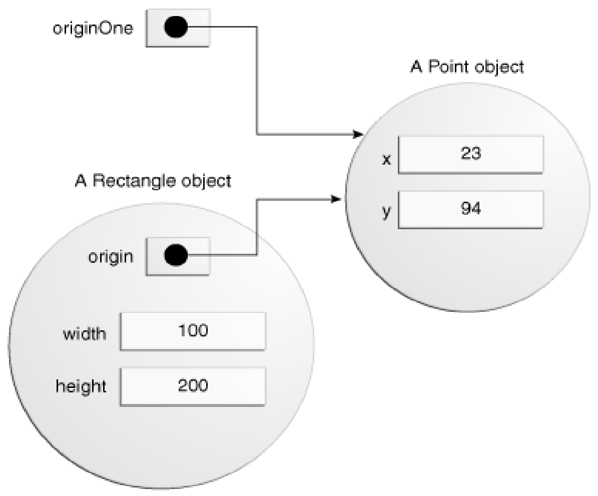
- This class contains a single constructor.
- You can recognize a constructor because its declaration uses the same name as the class and it has no return ty

Point originOne = new Point(23, 94);



Initializing an Object

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;
    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w; height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p; width = w; height = h;
    }
    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x; origin.y = y;
    }
    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
```



Rectangle rectOne = new Rectangle(originOne, 100, 200);

Initializing an Object

- All classes have at least one constructor.
- If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the *default constructor*.
- This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent.
- If the parent has no constructor (Object does have one), the compiler will reject the program.

Using Objects

40

Referencing an Object's Fields

- Object fields are accessed by their name.
- You may use a simple name for a field within its own class.
- For example, we can add a statement *within* the Rectangle class that prints the width and height:

```
System.out.println("Width and height are: " + width + ", " + height);
```

Referencing an Object's Fields

- Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

- For example, the code in the CreateObjectDemo class is outside the code for the Rectangle class. To access the width field within the Rectangle object, the code should be written as follows.

```
System.out.println("Width of rectOne: " + rectOne.width);
```

Calling an Object's Methods

objectReference.methodName(argumentList);

Example: int areaOfRectangle = new Rectangle(100, 50).getArea();

- Remember, invoking a method on a particular object is the same as sending a message to that object.
- In this case, the object that getArea() is invoked on is the rectangle returned by the constructor.

The Garbage Collector

- Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed.
- Managing memory explicitly is tedious and error-prone.
- The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them.
- The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

The Garbage Collector

- An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope.
- Or, you can explicitly drop an object reference by setting the variable to the special value null.
- Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.
- The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced.
- The garbage collector does its job automatically when it determines that the time is right.

More on Classes

46

Returning a Value from a Method

- A method returns to the code that invoked it when it
  - completes all the statements in the method,
  - reaches a return statement
  - throws an exception (covered later),

whichever occurs first.

- You declare a method's return type in its method declaration.
- Within the body of the method, you use the return statement to return the value.

Returning a Value from a Method

- Any method declared void doesn't return a value.
- It does not need to contain a return statement, but it may do so.
- In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

- If you try to return a value from a method that is declared void, you will get a compiler error.

Returning a Value from a Method

- Any method that is not declared void must contain a return statement with a corresponding return value, like this:

  ```
  return returnValue;
  ```

- The data type of the return value must match the method's declared return type;
- you can't return an integer value from a method declared to return a boolean.

Returning a Primitive Type

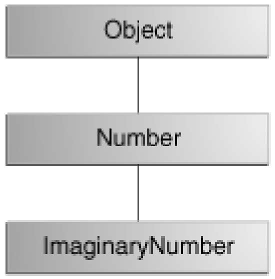
```
// a method for computing the area of the rectangle
public int getArea() {
        return width * height;
}
```

Returning a Reference Type

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                              Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;
}
```

Returning a Class or Interface

- When a method uses a class name as its return type, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type.



- Now suppose that you have a method declared to return a Number:

  public Number returnANumber() { ... }

  The returnANumber method can return an ImaginaryNumber but not an Object.

Returning a Class or Interface

- You can override a method and define it to return a subclass of the original method, like this:

```
public ImaginaryNumber returnANumber() { ... }
```

- This technique, called *covariant return type*, means that the return type is allowed to vary in the same direction as the subclass.
- **Note:** You also can use interface names as return types. In this case, the object returned must implement the specified interface.

Using the this Keyword

- Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called.
- You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

- The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument.
- To refer to the Point field **x**, the constructor must use **this.x**.

Using this with a Constructor

- From within a constructor, you can also use the this keyword to call another constructor in the same class.
- Doing so is called an *explicit constructor invocation*.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

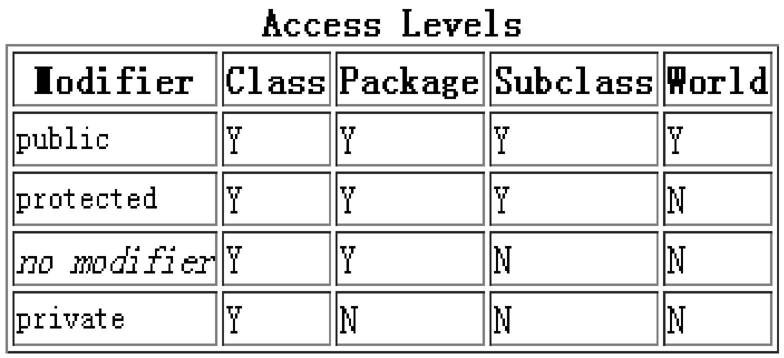
Controlling Access to Members of a Class

- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes)

Controlling Access to Members of a Class

- At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning.
- For members, there are two additional access modfiers: private and protected.
- The private modifier specifies that the member can only be accessed in its own class.
- The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

Controlling Access to Members of a Class



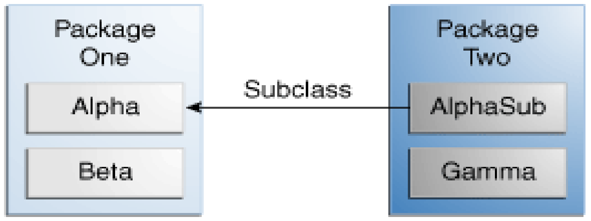
Access Levels

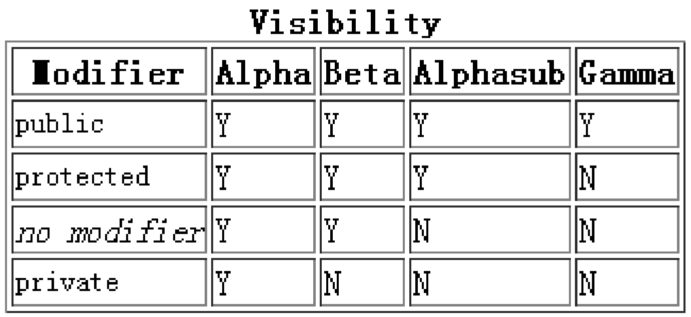
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Controlling Access to Members of a Class

- Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use.
- Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

The Visibility of the Members of the Alpha Class





Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Tips on Choosing an Access Level

- If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.
- Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.)
- Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Understanding Class Members

64

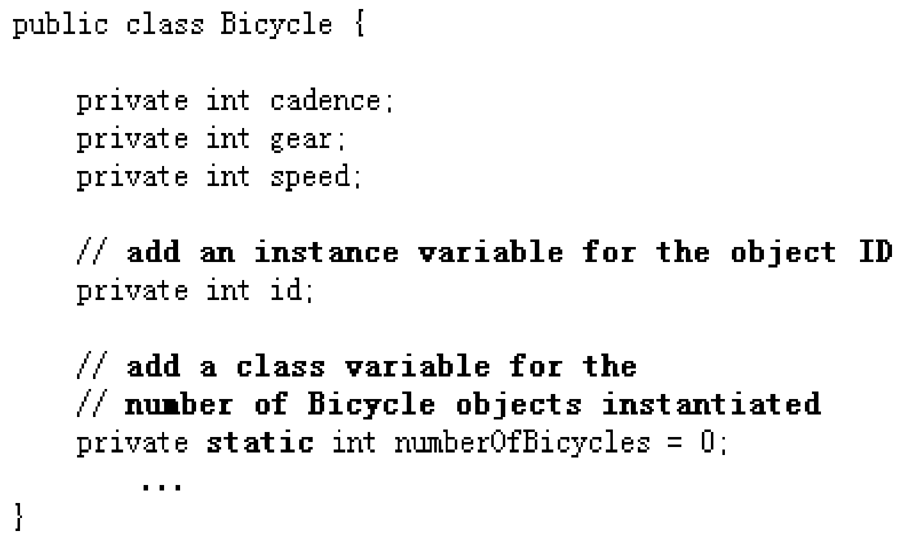
Class Variables

- When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*.
- In the case of the Bicycle class, the instance variables are cadence, gear, and speed. Each Bicycle object has its own values for these variables, stored in different memory locations.
- Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier.

Class Variables

- Fields that have the static modifier in their declaration are called *static fields* or *class variables*.
- They are associated with the class, rather than with any object.
- Every instance of the class shares a class variable, which is in one fixed location in memory.
- Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

Class Variables-Example



```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    // add an instance variable for the object ID
    private int id;

    // add a class variable for the
    // number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;
        ...
}
```

Class variables are referenced by the class name itself, as in
    Bicycle.numberOfBicycles

Note: You can also refer to static fields with an object reference like myBike.numberOfBicycles, but this is discouraged because it does not make it clear that they are class variables.

Class Methods (Static Methods)

- Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

```
ClassName.methodName(args)
```

- A common use for static methods is to access static fields. For example, we could add a static method to the Bicycleclass to access the numberOfBicycles static field:

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

Class Methods

- Not all combinations of instance and class variables and methods are allowed:
- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference.
- Class methods cannot use the this keyword as there is no instance for this to refer to.

Constants

- The static modifier, in combination with the final modifier, is also used to define constants.
- The final modifier indicates that the value of this field cannot change.

```
static final double PI = 3.141592653589793;
```

- Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so.
- By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_)

Initializing Fields

```
public class BedAndBreakfast {

    // initialize to 10
    public static int capacity = 10;

    // initialize to false
    private boolean full = false;
}
```

- This works well when the initialization value is available and the initialization can be put on one line.
- However, this form of initialization has limitations because of its simplicity.

Initializing Fields

- If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate.
- Instance variables can be initialized in constructors, where error handling or other logic can be used.
- To provide the same capability for class variables, the Java programming language includes *static initialization blocks*.

Static Initialization Blocks

```
static {
      // whatever code is needed for initialization goes here
 }

class Whatever {
     public static varType myVar = initializeClassVariable();
     private static varType initializeClassVariable() {
      // initialization code goes here
     }
}
```

Initializing Instance Members

- Normally, you would put code to initialize an instance variable in a constructor.
- There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.
- Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

```
{ // whatever code is needed for initialization goes here }
```

- The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

Summary of Creating and Using Classes and Objects

- A class declaration names the class and encloses the class body between braces.
- The class name can be preceded by modifiers.
- The class body contains fields, methods, and constructors for the class.
- A class uses fields to contain state information and uses methods to implement behavior.
- Constructors that initialize a new instance of a class use the name of the class and look like methods without a return type.

Summary of Creating and Using Classes and Objects

- You control access to classes and members in the same way: by using an access modifier such as public in their declaration.
- You specify a class variable or a class method by using the static keyword in the member's declaration.
- A member that is not declared as static is implicitly an instance member.
- Class variables are shared by all instances of a class and can be accessed through the class name as well as an instance reference.
- Instances of a class get their own copy of each instance variable, which must be accessed through an instance reference.

Summary of Creating and Using Classes and Objects

- You create an object from a class by using the new operator and a constructor. The new operator returns a reference to the object that was created.
- You can assign the reference to a variable or use it directly.
- Instance variables and methods that are accessible to code outside of the class that they are declared in can be referred to by using a qualified name. The qualified name of an instance variable looks like this:

  objectReference.variableName

Summary of Creating and Using Classes and Objects

- The qualified name of a method looks like this:

  objectReference.methodName(argumentList)

  or:

  objectReference.methodName()

- The garbage collector automatically cleans up unused objects. An object is unused if the program holds no more references to it.
- You can explicitly drop a reference by setting the variable holding the reference to null.

Nested Classes

```
class OuterClass {
        ...
        class NestedClass {
        ...
        }
}
```

Nested classes are divided into two categories: static and non-static.

- Nested classes that are declared static are called *static nested classes*.
- Non-static nested classes are called *inner classes*.

Nested Classes

```
class OuterClass {
        ...
        static class StaticNestedClass {
          ...
        }
        class InnerClass {
          ...
        }
}
```

- A nested class is a member of its enclosing class.
- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.
- As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- It can lead to more readable and maintainable code.

Static Nested Classes

- As with class methods and variables, a static nested class is associated with its outer class.
- And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.
- Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

Inner Classes

- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
- Because an inner class is associated with an instance, it cannot define any static members itself.
- Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Inner Classes

```
class OuterClass {
        ...
        class InnerClass {
            ...
        }
}
```

- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.
- To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Shadowing

```
public class ShadowTest {
        public int x = 0;
        class FirstLevel {
            public int x = 1;
            void methodInFirstLevel(int x) {
                  System.out.println("x = " + x);
                  System.out.println("this.x = " + this.x);
                  System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
            }
        }
        public static void main(String... args) {
            ShadowTest st = new ShadowTest();
            ShadowTest.FirstLevel fl = st.new FirstLevel();
            fl.methodInFirstLevel(23);
        }
}
```

Local Classes

- Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.
  - Declaring Local Classes
  - Accessing Members of an Enclosing Class
  - Shadowing and Local Classes
  - Local Classes Are Similar To Inner Classes

Declaring Local Classes

- You can define a local class inside any block.
- For example, you can define a local class in a method body, a for loop, or an if clause.

```
public class LocalClassExample {
    static String regularExpression = "[^0-9]";
    public static void validatePhoneNumber(
        String phoneNumber1, String phoneNumber2) {
        final int numberLength = 10;
        class PhoneNumber {
            String formattedPhoneNumber = null;
            PhoneNumber(String phoneNumber){
               String currentNumber = phoneNumber.replaceAll(
                 regularExpression, "");
               if (currentNumber.length() == numberLength)
                   formattedPhoneNumber = currentNumber;
               else
                   formattedPhoneNumber = null;
            }
            public String getNumber() {
                return formattedPhoneNumber;
            }
        }
        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);
        if (myNumber1.getNumber() == null)
            System.out.println("First nunber is invalid");
        else
            System.out.println("First nunber is " + myNumber1.getNumber());
        if (myNumber2.getNumber() == null)
            System.out.println("Second number is invalid");
        else
            System.out.println("Second number is " + myNumber2.getNumber());

    }
    public static void main(String... args) {
        validatePhoneNumber("123-456-7890", "456-7890");
    }

}
```

Accessing Members of an Enclosing Class

- A local class has access to the members of its enclosing class.
- In addition, a local class has access to local variables. However, a local class can only access local variables that are declared final.
- When a local class accesses a local variable or parameter of the enclosing block, it *captures* that variable or parameter.
  - For example, the PhoneNumber constructor can access the local variable numberLength because it is declared final; numberLength is a *captured variable*.
- However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or *effectively final*.

Local Classes Are Similar To Inner Classes

- Local classes are similar to inner classes because they cannot define or declare any static members.
- Local classes in static methods, such as the class PhoneNumber, which is defined in the static method validatePhoneNumber, can only refer to static members of the enclosing class.
  - For example, if you do not define the member variable regularExpresssion as static, then the Java compiler generates an error similar to "non-static variable regularExpression cannot be referenced from a static context."
- Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

Local Classes Are Similar To Inner Classes

- A local class can have static members provided that they are constant variables.
- A *constant variable* is a variable of primitive type or type String that is declared final and initialized with a compile-time constant expression.
- A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time.

```
public void sayGoodbyeInEnglish() {
    class EnglishGoodbye {
        public static final String farewell = "Bye bye";
        public void sayGoodbye() {
            System.out.println(farewell);
        }
    }
    EnglishGoodbye myEnglishGoodbye = new EnglishGoodbye();
    myEnglishGoodbye.sayGoodbye();
}
```

Anonymous Classes

- Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time.
- They are like local classes except that they do not have a name. Use them if you need to use a local class only once.
  - Declaring Anonymous Classes
  - Syntax of Anonymous Classes
  - Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class
  - Examples of Anonymous Classes

Declaring Anonymous Classes

- While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression.
- The following example, HelloWorldAnonymousClasses, uses anonymous classes in the initialization statements of the local variables frenchGreeting and spanishGreeting, but uses a local class for the initialization of the variable englishGreeting:

```
public class HelloWorldAnonymousClasses {
    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }
    public void sayHello() {
        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }
        HelloWorld englishGreeting = new EnglishGreeting();
        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };
        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }
    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
            new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

Syntax of Anonymous Classes

The anonymous class expression consists of the following:

- The new operator
- The name of an interface to implement or a class to extend.
  - In this example, the anonymous class is implementing the interface HelloWorld.
- Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression.
- Note: When you implement an interface, there is no constructor, so you use an empty pair of parentheses.
- A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Syntax of Anonymous Classes

- Because an anonymous class definition is an expression, it must be part of a statement.
- In this example, the anonymous class expression is part of the statement that instantiates the frenchGreeting object.
- This explains why there is a semicolon after the closing brace.

Accessing Local Variables of the Enclosing Scope

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
- Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.

Declaring and Accessing Members of the Anonymous Class

- Anonymous classes also have the same restrictions as local classes with respect to their members:
- You cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.
- You can declare the following in anonymous classes:

  Fields, Extra methods (even if they do not implement any methods of the supertype), Instance initializers, Local classes
- You cannot declare constructors in an anonymous class.

Enum Types

- An *enum type* is a special data type that enables for a variable to be a set of predefined constants.
- The variable must be equal to one of the values that have been predefined for it.
- Because they are constants, the names of an enum type's fields are in uppercase letters.
- In the Java programming language, you define an enum type by using the enum keyword.

```
public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                  THURSDAY, FRIDAY, SATURDAY }
```

- You should use enum types any time you need to represent a fixed set of constants.

Enum Types

```
public class EnumTest {
    Day day;
    public EnumTest(Day day) {
        this.day = day;
    }
    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;
            case FRIDAY:
                System.out.println("Fridays are better.");
                break;
            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;
            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }
    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

Enum Types

- Java programming language enum types are much more powerful than their counterparts in other languages.
- The enumdeclaration defines a *class* (called an *enum type*). The enum class body can include methods and other fields.
- The compiler automatically adds some special methods when it creates an enum. For example, they have a static values method that returns an array containing all of the values of the enum in the order they are declared.
- This method is commonly used in combination with the for-each construct to iterate over the values of an enum type.