Course of Programming in Java

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

The Java Tutorials

Object-Oriented Programming Concepts

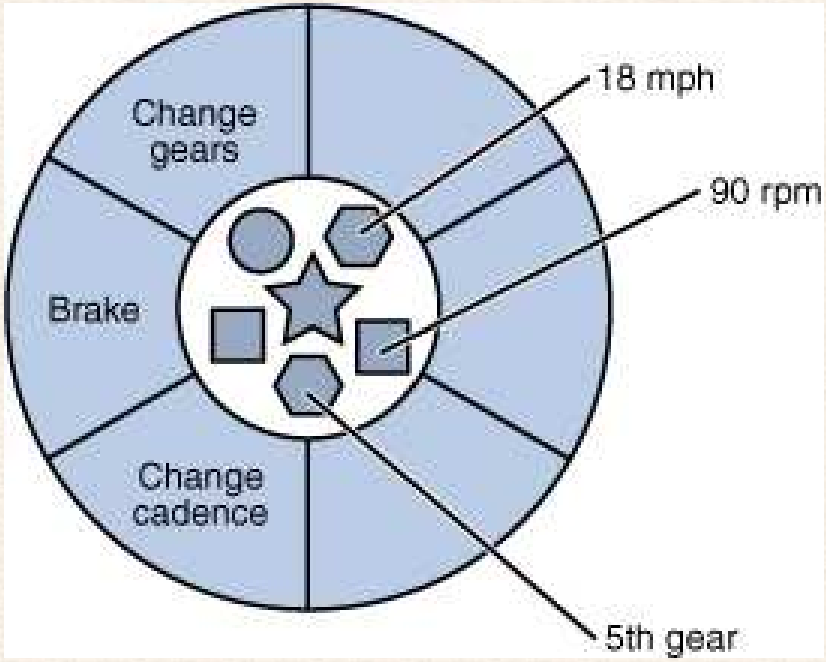
Language Basics

Classes and Objects

Interfaces and Inheritance

Object-Oriented Programming Concepts

Objects and Classes



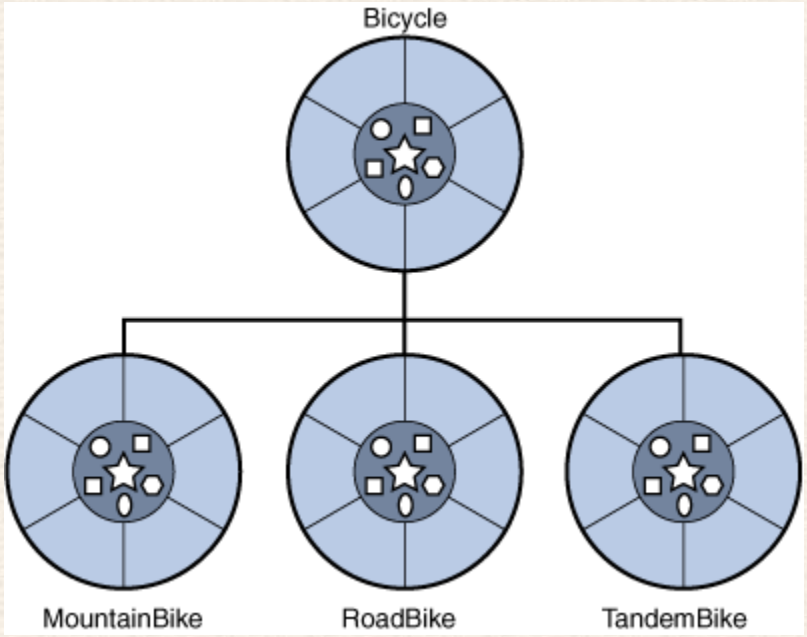
A bicycle modeled as a software object

- "What possible states can this object be in?"
- "What possible behavior can this object perform?"

```
class Bicycle {
        int cadence = 0;
        int speed = 0;
        int gear = 1;
        void changeCadence(int newValue) {
             cadence = newValue;
        }
        void changeGear(int newValue) {
             gear = newValue;
        }
        void speedUp(int increment) {
             speed = speed + increment;
        }

        void applyBrakes(int decrement) {
             speed = speed - decrement;
        }
        void printStates() {
             System.out.println("cadence:" + cadence + " speed:" + speed +
                                " gear:" + gear);
        }
}
```

```
class BicycleDemo {
     public static void main(String[] args) {
          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();
          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.speedUp(10);
          bike2.changeGear(3);
          bike2.printStates();
     }
}
```

Inheritance




```
class MountainBike extends Bicycle {
     // new fields and methods defining a mountain bike
}
```

take care to properly document the state and behavior that each superclass defines – that code will not appear in the source file of each subclass

Interface

Interfaces are used to provide contracts.

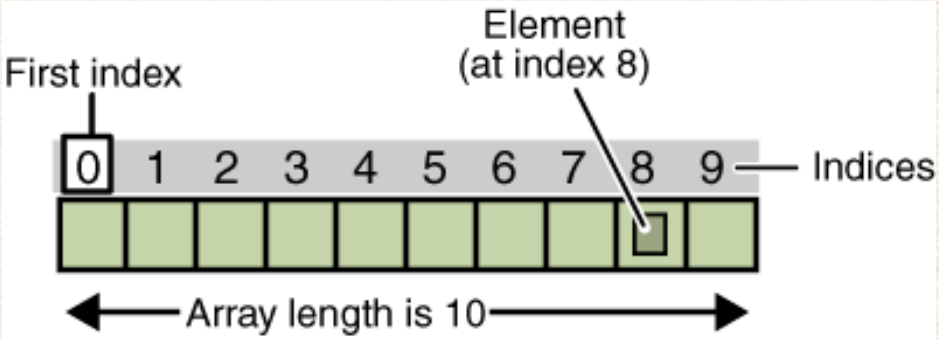
```
interface Bicycle {
        void changeCadence(int newValue);
        void changeGear(int newValue);
        void speedUp(int increment);
        void applyBrakes(int decrement);
}

class ACMEBicycle implements
 Bicycle {
   // remainder of this class implemented as before
}
```

Language Basics

Variables

- **Instance Variables (Non-Static Fields)** Their values are unique to each *instance* of a class (to each object); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** There is exactly one copy of this variable in existence. The number of gears for a particular kind of bicycle: `static int numGears = 6;` (The keyword `final` could be added to indicate that the number of gears will never change.)
- **Local Variables** A method will often store its temporary state in *local variables*: e.g. `int count = 0;` between the opening and closing braces of a method. Not accessible from the rest of the class.
- **Parameters** In `public static void main(String[] args)` the `args` variable is the parameter to `main` method.

Arrays




```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                            {"Smith", "Jones"}};
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones
    }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n',
                            'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

The output from this program is:

```
caffein
```

Statements

```
void applyBrakes(){
      if (isMoving) currentSpeed--;
}
```

```
class IfElseDemo {
    public static void main(String[] args) {
        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

```
class SwitchDemo2 {
    public static void main(String[] args) {
        int month = 2;
        int year = 2000;
        int numDays = 0;
        switch (month) {
            case 1:  case 3:  case 5:  case 7:
            case 8:  case 10: case 12:
                numDays = 31;
                break;
            case 4:  case 6:  case 9:  case 11:
                numDays = 30;
                break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                     || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

```
public class StringSwitchDemo {
    public static int getMonthNumber(String month) {
        int monthNumber = 0;
        if (month == null) { return monthNumber; }
        switch (month.toLowerCase()) {
            case "january":    monthNumber =  1; break;
            case "february":   monthNumber =  2; break;
            case "march":      monthNumber =  3; break;
            case "april":      monthNumber =  4; break;
            case "may":        monthNumber =  5; break;
            case "june":       monthNumber =  6; break;
            case "july":       monthNumber =  7; break;
            case "august":     monthNumber =  8; break;
            case "september":  monthNumber =  9; break;
            case "october":    monthNumber = 10; break;
            case "november":   monthNumber = 11; break;
            case "december":   monthNumber = 12; break;
            default:           monthNumber =  0; break;
        }
        return monthNumber;
    }
    public static void main(String[] args) {
        String month = "August";
        int returnedMonthNumber = StringSwitchDemo.getMonthNumber(month);
        if (returnedMonthNumber == 0) System.out.println("Invalid month");
        else  System.out.println(returnedMonthNumber);
    }}
```

```
class EnhancedForDemo {
     public static void main(String[] args){
          int[] numbers = {1,2,3,4,5,6,7,8,9,10};
          for (int item : numbers) {
            System.out.println("Count is: " + item);
          }
     }
}
```

Questions

Questions: variables

1. The term "instance variable" is another name for ____.
2. The term "class variable" is another name for ____.
3. A local variable stores temporary state; it is declared inside a ____.
4. A variable declared within the opening and closing parenthesis of a method signature is called a _____.
5. What are the eight primitive data types supported by the Java programming language?
6. Character strings are represented by the class ____.
7. An ____ is a container object that holds a fixed number of values of a single type.

Questions: operators

1. Consider the following code snippet.

   ```
   arrayOfInts[j] > arrayOfInts[j+1]
   ```

   Which operators does the code contain?

2. Consider the following code snippet.

   ```
   int i = 10;
   int n = i++%5;
   ```

   a. What are the values of `i` and `n` after the code is executed?

   b. What are the final values of `i` and `n` if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i`))?

3. To invert the value of a `boolean`, which operator would you use?

4. Which operator is used to compare two values, = or == ?

5. Explain the following code sample: `result = someCondition ? value1 : value2;`

Questions: expressions, statements

1. Operators may be used in building ____, which compute values.
2. Expressions are the core components of ____.
3. Statements may be grouped into ____.
4. The following code snippet is an example of a ____ expression.

```
1 * 2 * 3
```

5. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a ____.
6. A block is a group of zero or more statements between balanced ____ and can be used anywhere a single statement is allowed.

Questions: control flow statements

1. The most basic control flow statement supported by the Java programming language is the `___` statement.
2. The `___` statement allows for any number of possible execution paths.
3. The `___` statement is similar to the `while` statement, but evaluates its expression at the `___` of the loop.
4. How do you write an infinite loop using the `for` statement?
5. How do you write an infinite loop using the `while` statement?

Classes

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

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

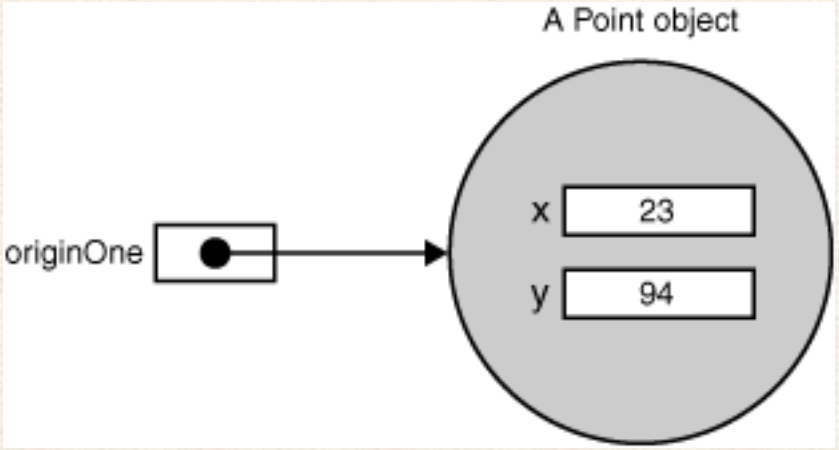
```
public class MountainBike extends Bicycle {
    // has one field
    public int seatHeight;

    // has one constructor
    public MountainBike(int startHeight, int startCadence, int
startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    // has one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

Creating Objects

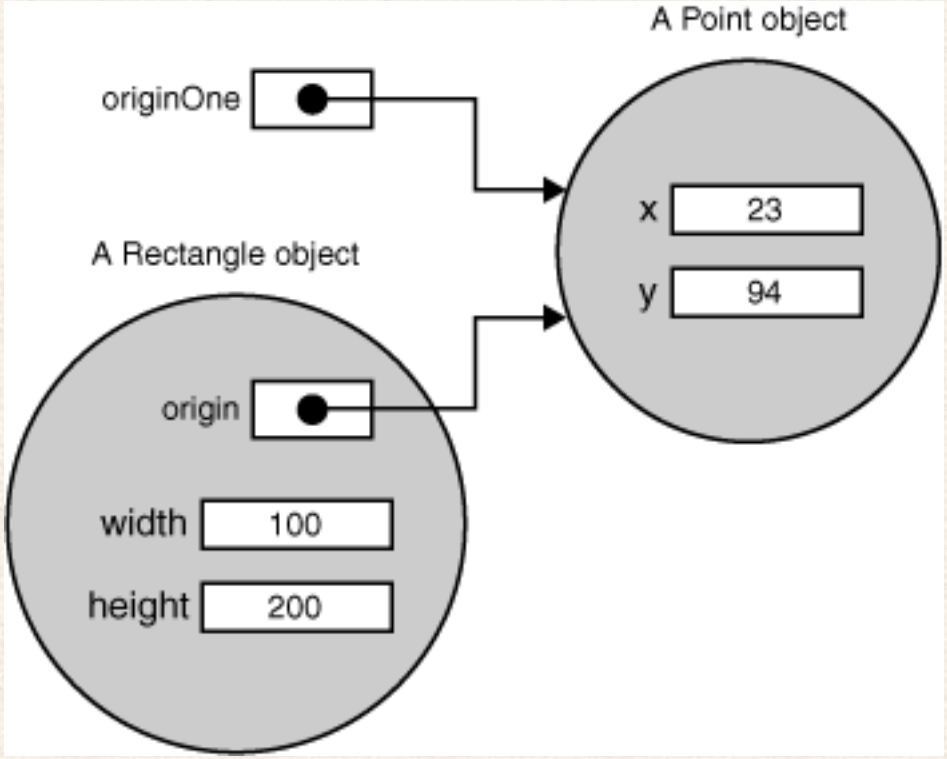
Instantiating a class means the same thing as *creating an object* – the object needs to be *initialized* afterwards by initialization blocks and the constructor.

```
Point originOne;
```


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



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




```
System.out.println("Width of rectOne: " + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
int height = new Rectangle().height;
System.out.println("Area of rectOne: " + rectOne.getArea());
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Explicit constructor invocation

```
public class Rectangle {
    private int x, y;
    private int width, height;
    public Rectangle() {
        this(0, 0, 0, 0);
    }
    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

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

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`.

Class Members (i.e. static)

```
public class Bicycle{
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;
    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
        // increment number of Bicycles and assign ID number
        id = ++numberOfBicycles;
    }
    // new method to return the ID instance variable
    public int getID() { return id; }
    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    } ...
}
```

Initializing Class Members

Either:

```
class Whatever {
    public static varType myVar[] = new varType[7];
    static {
        //initialization code goes here
    }
}
```

or:

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

Initializing Instance Members

Either:

```
class Whatever {
    public varType myVar[] = new varType[7];
    {
        //initialization code goes here
    }
}
```

or:

```
class Whatever {
    public varType myVar[] = initializeMemberVar();
    protected final varType initializeMemberVar() {
        //initialization code goes here
    }
}
```

Annotations

Annotations provide data about a program that is not part of the program.

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { }
...
@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

If there is just one element named "value," then the name may be omitted:

```
@SuppressWarnings("unchecked")
void myMethod() { }
```

If an annotation has no elements, the parentheses may be omitted:

```
@Override
void mySuperMethod() { }
```

Questions: Classes and Objects

1. Consider the following class:

```
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}
```

   a. What are the class variables?

   b. What are the instance variables?

   c. What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5; b.y = 6;
a.x = 1; b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

2. What's wrong with the following program?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
        Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

3. The following code creates one array and one string object. How many references to those objects exist after the code executes? Is either object eligible for garbage collection?

```
...
String[] students = new String[10];
String studentName = "Peter Parker";
students[0] = studentName;
studentName = null;
...
```

4. How does a program destroy an object that it creates?

5. What is wrong with the following interface?

```
public interface House {
    @Deprecated
    void open();
    void openFrontDoor();
    void openBackDoor();
}
```

6. Consider this implementation of the House interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler complains that open has been deprecated (in the interface). What can you do to get rid of that warning?

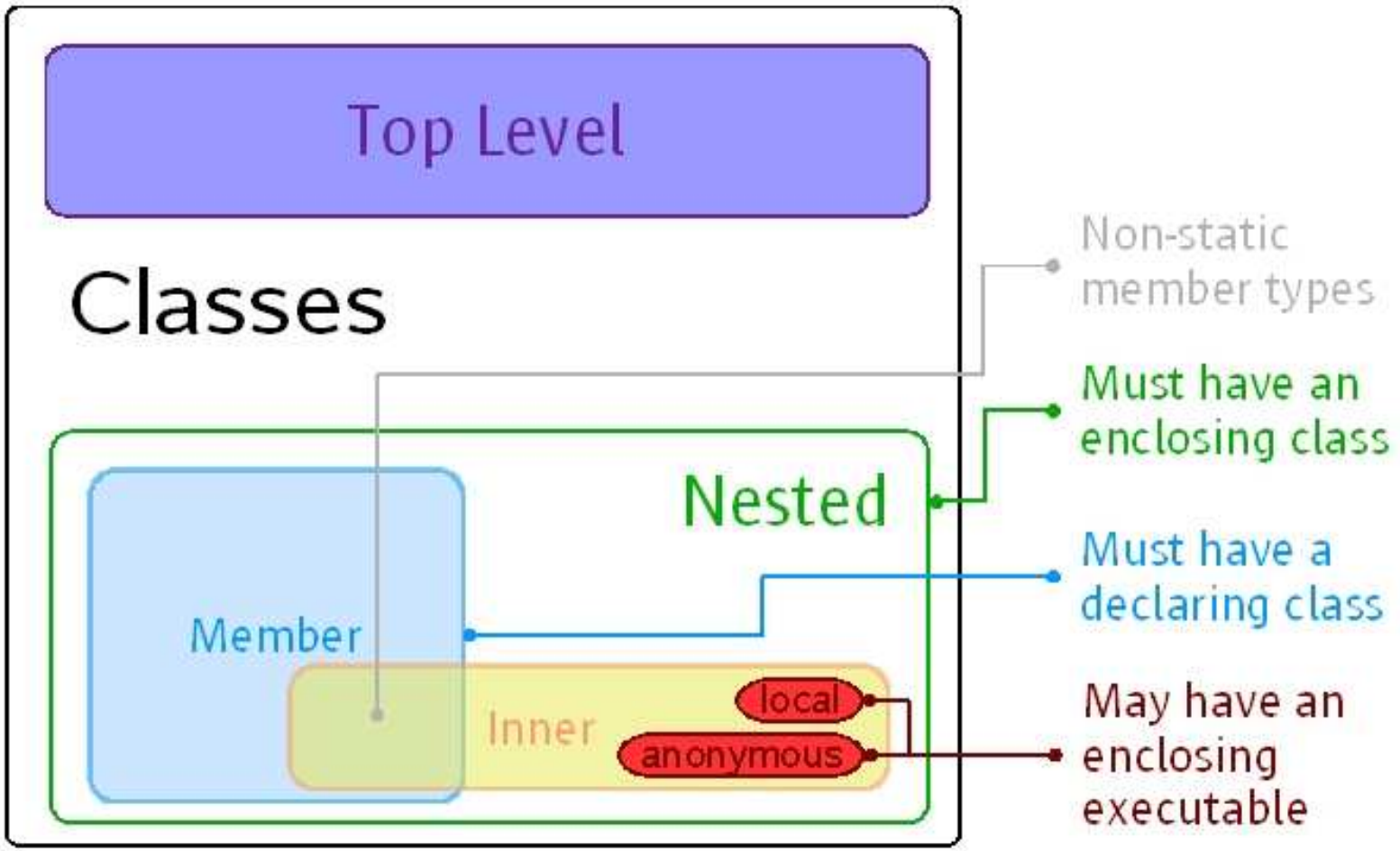
Nested Classes

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

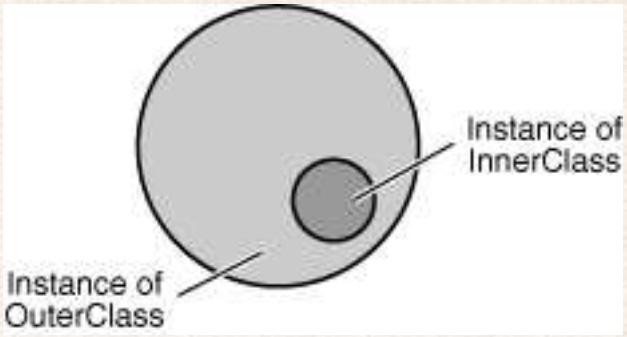
As a member of the `OuterClass`, a nested class can be declared `private`, `public`, `protected`, or *package private*. (Recall that outer classes can only be declared `public` or *package private*.)

```
OuterClass.StaticNestedClass nestedObject =
    new OuterClass.StaticNestedClass();
```

Taxonomy of Classes in the Java Programming Language



Inner Classes



An instance of `InnerClass` has direct access to the methods and fields of its enclosing instance of `OuterClass`.

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

```
public class DataStructure {
    //create an array
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public DataStructure() {
        //fill the array with ascending ints
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }

    public void printEven() {
        //print out values of even indices
        InnerEvenIterator iterator =
            this.new InnerEvenIterator();
        while (iterator.hasNext()) {
            System.out.println(
                iterator.getNext() + " ");
        }
    }
    //inner class implements
    // the Iterator pattern
    private class InnerEvenIterator {
        //start stepping from the beginning
        private int next = 0;

        public boolean hasNext() {
            //check if the current element
            //    is the last in the array
            return (next <= SIZE - 1);
        }

        public int getNext() {
            //record a value of an even index
            int retValue = arrayOfInts[next];
            //get the next even element
            next += 2;
            return retValue;
        }
    }


    public static void main(String s[]) {
       //fill the array with integer values
       //print out only values of even indices
       DataStructure ds = new DataStructure();
       ds.printEven();
    }
}
```

Enums

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass; // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass()   { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant
    public static final double G = 6.673E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
```

```
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println(
          "Usage:  java Planet <weight>");
            System.exit(-1);
        }

        double earthWeight =
            Double.parseDouble(args[0]);

        double mass =
          earthWeight/EARTH.surfaceGravity();

        for (Planet p : Planet.values())
           System.out.printf(
               "Your weight on %s is %f%n", p,
               p.surfaceWeight(mass));
    }
}
```

Questions

1. The program `Problem.java` doesn't compile. What do you need to do to make it compile? Why?

```
public class Problem {
        String s;
        static class Inner {
                void testMethod() {
                   s = "Set from Inner";
                }
        }
}
```

2. Use the Java API documentation for the `Box` class (in the `javax.swing` package) to help you answer the following questions.

   a. What static nested class does `Box` define?

   b. What inner class does `Box` define?

   c. What is the superclass of `Box`'s inner class?

   d. Which of `Box`'s nested classes can you use from any class?

   e. How do you create an instance of `Box`'s `Filler` class?

Interfaces

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures
   int turn(Direction direction,   // An enum with values RIGHT, LEFT
            double radius, double startSpeed, double endSpeed);
   int changeLanes(Direction direction, double startSpeed, double endSpeed);
   int signalTurn(Direction direction, boolean signalOn);
   int getRadarFront(double distanceToCar, double speedOfCar);
   int getRadarRear(double distanceToCar, double speedOfCar);
         ......
   // more method signatures
}
```

Interfaces can be *implemented* by classes or *extended* by other interfaces.

```
public class OperateBMW760i implements OperateCar {
   // the OperateCar method signatures, with implementation
   int signalTurn(Direction direction, boolean signalOn) {
      //code to turn BMW's LEFT/RIGHT turn indicator lights on/off
   }
   // other members, as needed -- for example, helper classes
   // not visible to clients of the interface
}
```

Uses of interfaces:

- The robotic car example shows an interface being used as an **industry standard** *Application Programming Interface (API)*.
- APIs are also common in **commercial software products**. E.g. a package of digital image processing methods that are sold to companies making end-user graphics programs.
- Interfaces allow **multiple inheritance** by (for example) forwarding.
- Interfaces simplify the use of "plug-in style" alternative implementations.

While a class can extend only a single class, an interface can extend, and a class can implement, multiple interfaces.

```
public interface Relatable {

   // this (object calling isLargerThan) and
   // other must be instances of the same
class
   // returns 1, 0, -1 if this is greater
   // than, equal to, or less than other
   public int isLargerThan(Relatable other);
}
```

```
public class RectanglePlus implements
Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(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;
    }

    // a method required to implement the
    // Relatable interface
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect =
            (RectanglePlus)other;

        if (this.getArea() >
                otherRect.getArea())
            return 1;
        else return 0;
    }
}
```

Inheritance

A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

```
public class MountainBike extends Bicycle {
    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int
startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

- using a subtype in context of a supertype

```
Bicycle bike = new MountainBike(); // implicit cast
if (bike instanceof MountainBike) {
   MountainBike myBike = (MountainBike)obj; //explicit
cast
}
```

- An overriding method can return a subtype of the type returned by the overridden method (*covariant return type*).
- Using the `@Override` annotation, if the compiler detects that the method does not exist in one of the superclasses, it will generate an error.
- Defn. a method with the same signature as a method in a superclass:

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates compile-time error
Subclass Static Method	Generates compile-time error	Hides

Another MountainBike:

```
public class MountainBike extends Bicycle{
  private String suspension;

  public MountainBike(int startCadence, int startSpeed, int startGear, String
suspensionType){
    super(startCadence, startSpeed, startGear);
    this.setSuspension(suspensionType);
  }

  public String getSuspension(){
    return this.suspension;
  }

  public void setSuspension(String suspensionType){
    this.suspension = suspensionType;
  }

  public void printDescription(){
    super.printDescription();
    System.out.println("The MountainBike has a " + getSuspension()
            + " suspension.");
  }
}
```

Another subclass of `Bicycle`:

```
public class RoadBike extends Bicycle{
  private int tireWidth; // In millimeters (mm)

  public RoadBike(int startCadence, int startSpeed, int startGear, int
newTireWidth){
    super(startCadence, startSpeed, startGear);
    this.setTireWidth(newTireWidth);
  }

  public int getTireWidth(){
    return this.tireWidth;
  }

  public void setTireWidth(int newTireWidth){
    this.tireWidth = newTireWidth;
  }

  public void printDescription(){
    super.printDescription();
    System.out.println("The RoadBike has " + getTireWidth()
            + " MM tires.");
  }
}
```

Recall also how `super` was used in `MountainBike`'s constructor.

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();

  }
}
```

The following is the output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

The Keyword `super`

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Here is a subclass, called `Subclass`, that overrides `printMethod()`:

```
public class Subclass extends Superclass {

    public void printMethod() { //overrides printMethod in Superclass
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

`Object` as a Superclass

- `protected Object clone() throws CloneNotSupportedException`
  Creates and returns a copy of this object.
- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable`
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- `public final Class getClass()`
  Returns the runtime class of an object.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

Final Classes and Methods

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...

    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Abstract Methods and Classes

An *abstract method* is a method that is declared without an implementation:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, or inherits abstract methods and does not implement (override) them, the class itself *must* be declared `abstract`:

```
public abstract class GraphicObject {
   // declare fields
   // declare non-abstract methods
   abstract void draw();
}
```

A class can implement an interface partially by being abstract.

```
abstract class X implements Y {
  // implements all but one method of Y
}
class XX extends X {
  // implements the remaining method in Y
}
```

Abstract classes provide part of functionality, the rest provided by subclasses:

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

Questions

1. What methods would a class that implements the `java.lang.CharSequence` interface have to implement?
2. What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue){
        System.out.println("Hi Mom");
    }
}
```

3. Fix the interface in question 2.
4. Is the following interface valid?

```
public interface Marker {
}
```

5. Consider the following two classes:

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

a. Which method overrides a method in the superclass?

b. Which method hides a method in the superclass?

c. What do the other methods do?

6. Consider the `Card`, `Deck`, and `DisplayDeck` classes. What `Object` methods should each of these classes override?