PROGRAMMING IN

Overview

- Literals
- Brief examples of Java applications
- OOP!

Announcements

Course website: http://sipb.mit.edu/iap/java/

Email: sipb-iap-java@mit.edu

Wake up and smell the coffee!

Software

Java Development Kit (JDK) - http://java.sun.com/javase/downloads/index.jsp

Eclipse Platform - http://www.eclipse.org/

Reference

The Java Tutorial - http://java.sun.com/docs/books/tutorial/index.html

Java Language API - http://java.sun.com/javase/reference/api.jsp

Java SE Documentation - http://java.sun.com/javase/downloads/index.jsp

Java SE Source Code - http://java.sun.com/javase/downloads/index.jsp

Where you'll find Java [incomplete list]

- Portable devices (cell phones, PDAs…)
- UROPs
- Mars (NASA Rovers)
- Eclipse
- Banking apps

Literals (Compiler-time values)

Booleans:	true, false
Numbers:	1, 2.2, 5e-20, 200L, 0x00FF, 0800
Characters:	'c', '\u0108', '\n'
Strings:	"Java!"
Class:	Object.class
Special:	null

OBJECT-ORIENTED

PROGRAMMING

```
package tvald.intro2java;
/**
 * This is a demo program.
 * @author tvald
 * @date 1/5/2009
 */
public class Hello {

    public static void main(String[] args) {

        System.out.println("Mmmm... Smell that coffee!"); // say hi

    }

}
```

What is a package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage.

```
package edu.mit.sipb.iap;

import java.util.*;     // wildcard import (load as needed)
import java.lang.Math; // import constant (static import)

public class Test {

    public static void main(String[] args){
        double x = Math.E;
        double rad = 2 * Math.PI;

        LinkedList list1;  // java.util.LinkedList
        ArrayList list2;   // java.util.ArrayList
    }
}
```

What is a class?

A class is a structure for organizing classes and interfaces in a logical manner. Placing your code into classes makes repetition and variation of functionality easier to manage.

```
package tvald.geom;

public class Point {
    double x, y;
    public Point(double x, double y){
        this.x = x; this.y = y;
    }
    public void translate(double dx, double dy){
        x += dx; y += dy;
    }
    public void rotate(double cx, double cy, double theta){
        // implementation omitted ...
    }
}
```

```
// elsewhere
Point p = new Point(1,1);
p.rotate(0,0,.5);
```

A bit of philosophy…

Plato

5th-4th century B.C.

Athens, Greece

...there is a form for every object or quality in reality: forms of dogs, human beings, mountains, colors, courage, love, and goodness. Form answers the question "what is that?" ...the object was essentially or "really" the Form and that the phenomena were mere shadows mimicking the Form; that is, momentary portrayals of the Form under different circumstances. The problem of universals - how can one thing in general be many things in particular - was solved by presuming that Form was a distinct singular thing but caused plural representations of itself in particular objects.

http://en.wikipedia.org/wiki/Theory_of_Forms

What is a Class?

A class is a blueprint or prototype from which objects are created.

```
<modifiers> class name {

// members: fields, methods, constructors

}
```

Convention:

Class names begin with an uppercase letter
Use “camel case” - `FirstSecondThird`.
Use only letters.

What is a Class?

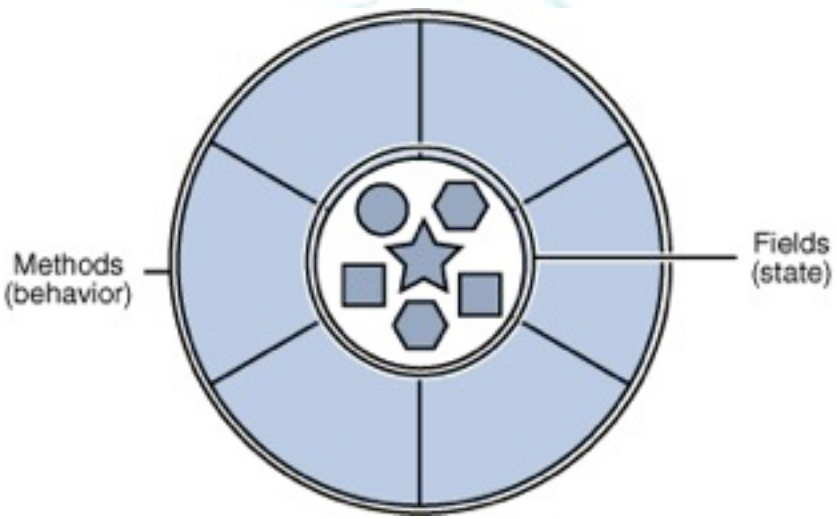
```
public class Hello {

    public static String text;

    public static void sayHi(){
        System.out.println("Mmmm... Smell that " + text + "!");
    }

    public static void main(String[] args) {
        text = "coffee";
        sayHi();
    }

}
```

What is an Object?

```
public class Hello {

    String text;

    public Hello(String s){
        text = s;
    }

    public void sayHi(){
        System.out.println("Mmmm... Smell that " + text + "!");
    }

    public static void main(String[] args) {
        Hello h = new Hello("coffee");
        h.sayHi();
    }

}
```

What is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life.

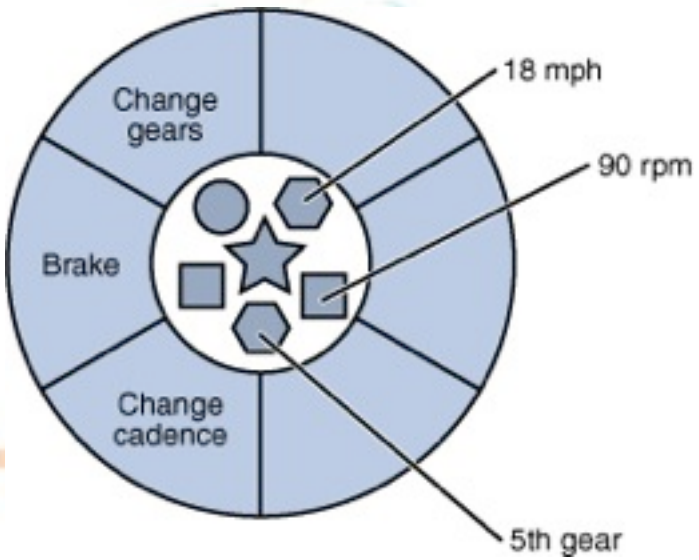


Software object

Image from The Java Tutorial

What is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life.



Bicycle modeled as software object

Image from The Java Tutorial

```
public class Bicycle {

    // the Bicycle class has two fields
    public int gear;
    public int speed;

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

    // the Bicycle class has three methods
    public void setGear(int newValue) {
        gear = newValue;
    }

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

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

Constructors

```
class Bicycle {

    public Bicycle(int startSpeed, int startGear) {
        gear = startGear;
        speed = startSpeed;
    }
    public Bicycle() {
        gear = 1;
        speed = 0;
    }

    /*
     * Omit the rest of the class definition
     */

    public static void main(String[] args) {
        Bicycle myBike = new Bicycle(0, 8);
        Bicycle otherBike = new Bicycle();
    }
}
```

Constructors

MUST have same name as class
MUST NOT have a return type
If no constructor is supplied, the compiler creates a default constructor to initialize instance members.

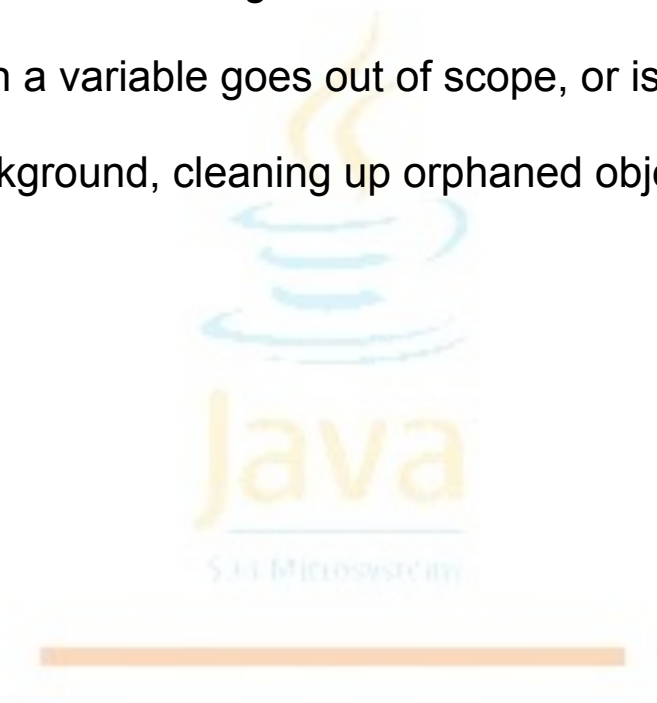
```
modifiers classname (parameter list) throws exceptionlist {
    // method body
}
```

Taking out the garbage

The Java VM includes an automatic Garbage Collector!

References are dropped when a variable goes out of scope, or is set to something else.

Garbage collector runs in background, cleaning up orphaned objects, freeing up memory.



Variable types

The type of variable is determined entirely by the location of its declaration plus the static keyword.

Instance field – non-static
Class field – static
Local variables & parameters – within method body / declaration

```
class TypesDemo {

    int instanceVar;

    static int classVar;

    public static void main(String[] parameter) {
        int localVar;
    }
}
```

Variable types

The type of variable is determined entirely by the location of its declaration plus the static keyword.

Instance field – non-static
Class field – static
Local variables & parameters – within method body / declaration

```
class TypesDemo {

    int instanceVar;

    static int classVar;

    public static void main(String[] parameter) {
        int localVar;
    }
}
```

Resolving variables

1. Local variables and parameters share a namespace.
2. Instance and class variables share a namespace.
3. Static imports exist in a file-wide namespace.

```
import static java.lang.Math.PI;

class TypesDemo {

    int instanceVar;

    static int classVar;

    public static void main(String[] parameter) {
        int localVar;
    }
}
```

Resolving variables

1. Local variables and parameters share a namespace.
2. Instance and class variables share a namespace.
3. Static imports exist in a file-wide namespace.

```
import static java.lang.Math.PI;

class TypesDemo {

    int instanceVar;

    static int classVar;

    public static void main(String[] parameter) {
        int localVar;
    }
}
```

Resolving variables

1. Local variables and parameters share a namespace.
2. Instance and class variables share a namespace.
3. Static imports exist in a file-wide namespace.

```
import static java.lang.Math.PI;

class TypesDemo {

    int instanceVar;

    static int classVar;

    public static void main(String[] parameter) {
        int localVar;
    }
}
```

Look at `this`

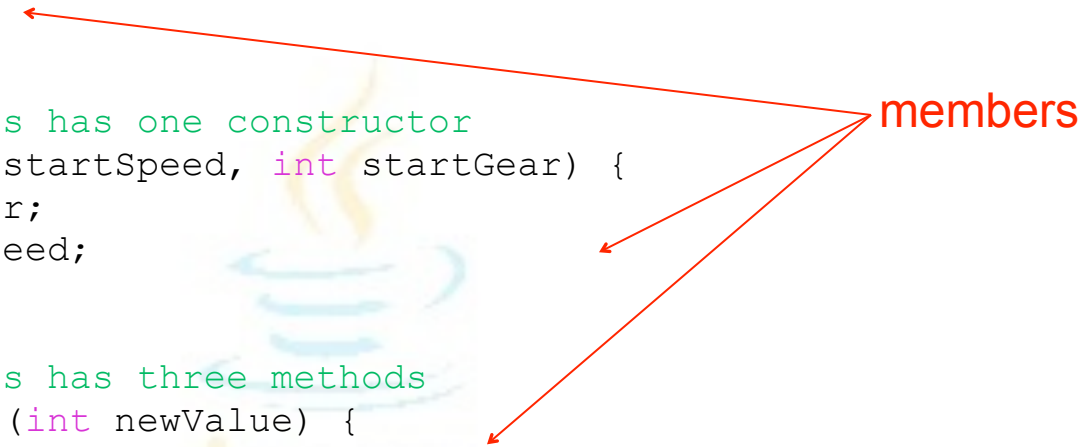
Local variables and parameters can *shadow* class fields.

```
public MyClass {
      int variable = 0;
      public int doMath(int variable){
          return variable + this.variable; // instance variable
      }
}
```

This can be used to call constructors from the first line of another constructor.

```
public MyClass {
     public MyClass() {
          this(2);
     }
     public MyClass(int i) { ... }
}
```

```
public class Bicycle {

    // the Bicycle class has two fields
    public int gear;
    public int speed;

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

    // the Bicycle class has three methods
    public void setGear(int newValue) {
        gear = newValue;
    }

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

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



Default Initialization (fields only)

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Static Initializers

```
public class Test {

// A field or static field may be initialized at it's declaration:
    public static final int NORTH = 1;
    private int direction = NORTH;

// A static initialization block can execute statements
// when the class is first loaded:
    public static int[] fibonacci;
    static {
        fibonacci = new int[10]
        fibonacci[1] = 1;
        for (int i = 2; i < fibonacci.length; i++)
            fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
    }

// Or, you could use a private static method (which could
// be reused later, if necessary):
    public static int myVar = initializeClassVariable();
    private static int initializeClassVariable() {
        //initialization code goes here
    }
}
```

Variable modifiers

```
// scope (applies only to fields)
public    int i; // class, package, subclass, world
protected int i; // class, package, subclass
          int i; // class, package
private   int i; // class


// other
final int CONSTANT_VALUE; // fixed value
public static final int
         UP = 1,
         DOWN = 2; // can be used in a similar fashion to #DEFINE
```

Convention:

Constants use only uppercase letters with words separated by underscores.

Object-Oriented Programming

Java is very Object-Oriented. Even primitives can be wrapped in Objects.

Objects can have a lifetime greater than the object that created them.

An Object-Oriented language should support:

- *Encapsulation* - information hiding and modularity (abstraction)
- *Polymorphism* - behavior is dependent on the nature of the object receiving a message
- *Inheritance* - new classes are defined based on existing classes to obtain code re-use and organization
- *Dynamic binding* - objects could come from anywhere, possibly across the network. Send messages to objects without knowing their specific type at the time you write your code.

What is an Object?

Bundling code into software objects provides a number of benefits, including:

Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Encapsulation

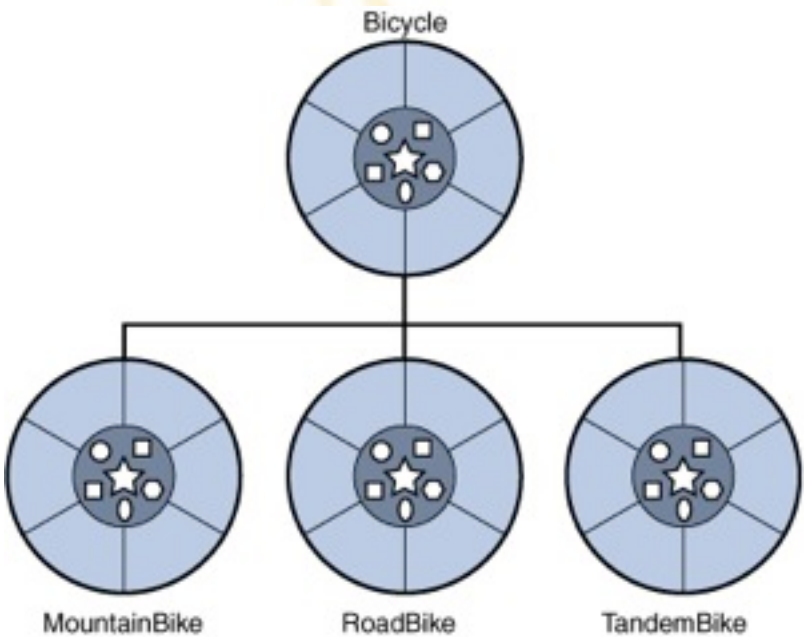
```
public class Bicycle {

    private int gear;
    private int speed;

    public int getGear() {  return gear; }

    public void setGear(int newValue) {
        if (gear > 0) gear = newValue;
    }

    public int getSpeed() {  return speed; }

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

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

More Constructors

```
public class NoninstantiableClass {

    private NoninstantiableClass() {
        // this can't be instantiated by anybody else
    }

    public static NoninstantiableClass factoryMethod() {
        return new NoninstantiableClass();
    }
}
```

What is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software.



A hierarchy of bicycles

What is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software.

```
class MountainBike extends Bicycle {

     // new fields and methods defining a mountain bike would go here

}
```

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

What is Inheritance?

```
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 startSpeed, int startGear)
    {
        speed = startSpeed
        gear = startGear;
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

Overriding methods and fields

```
public class Cat extends Animal {
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
    }
    public static void main(String[] args) {
        Animal myAnimal = new Cat();
        myAnimal.testInstanceMethod();
    }
}
class Animal {
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```

Overriding methods and fields

```
public class Cat extends Animal {
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
    }
    public static void main(String[] args) {
        Animal myAnimal = new Cat();
        myAnimal.testInstanceMethod();
    }
}
class Animal {
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```

```
The instance method in Cat.
```

Super

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

Super

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

```
Printed in Superclass.
Printed in Subclass
```

Super constructors

```
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 startSpeed, int startGear)
    {
        speed = startSpeed
        gear = startGear;
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

Super constructors

```
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 startSpeed, int startGear)
    {
        super(startSpeed, startGear);

        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

Object references and casting

```
// primitives
int i = 0;
int j = i;
j = 1;
System.out.println("i = " + i); // primatives=>i=0,Analogous objects=>i=1

MountainBike mountainBike = new MountainBike();
Bicycle bike = mountainBike; // automatically casted
Object o = bike; // all classes descend from Object

o = new MountainBike();
if (o instanceof MountainBike) {
        mountainBike = (MountainBike)o; // cast it back
}

o = new Object();
mountainBike = (MountainBike)o; // ClassCastException at runtime
```

Final

A *final class* is a class cannot be subclassed.

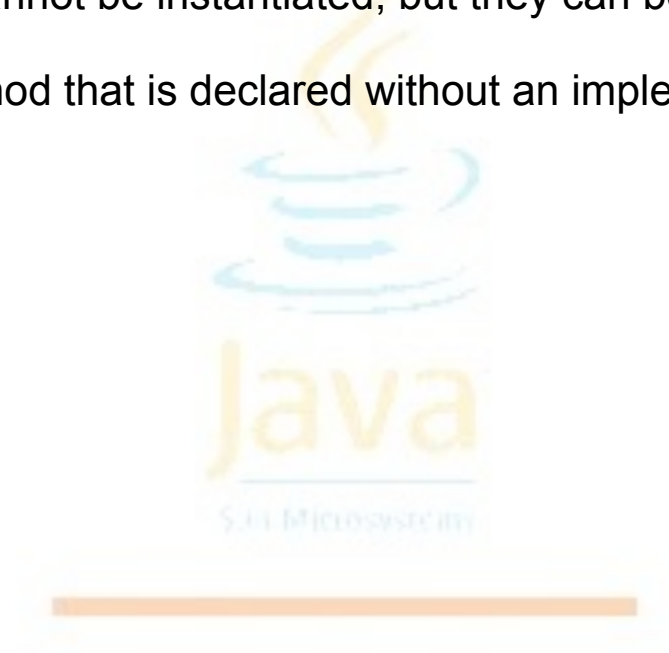
A *final method* is a method that cannot be overridden.

```
public final class String {
    // ...
}


public class Object {
    public final Class getClass(){
        // ...
    }
}
```

Abstract

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

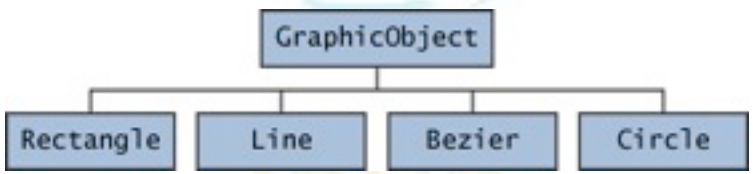
An *abstract method* is a method that is declared without an implementation



Abstract

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

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




```
abstract class GraphicObject {
    int x, y;
    void moveTo(int newX, int newY) {
        //...
    }
    abstract void draw();
}
```

What is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface.

```
interface Bicycle {
    // constant declarations, if any
    public static final MAX_GEAR = 10;

    void changeGear(int newValue); //interfaces are completely abstract
    void speedUp(int increment);
    public abstract void applyBrakes(int decrement); // implied
}


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

Interface Types

Interface types work just like class types.

```
public interface Relatable {
    /**
      * this (the 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 boolean isEqual(Relatable rel1, Relatable rel2) {
        if (rel1.isLargerThan(rel2) == 0) return true;
        else return false;
}
```

Extending interfaces

```
public interface DoIt {
    void doSomething(int i, double x);
    int doSomethingElse(String s);
}

public interface DoItPlus extends DoIt {
    boolean didItWork(int i, double x, String s);
}
```