Session 7

Methods

Strings

Constructors

this

Inheritance

Java Methods

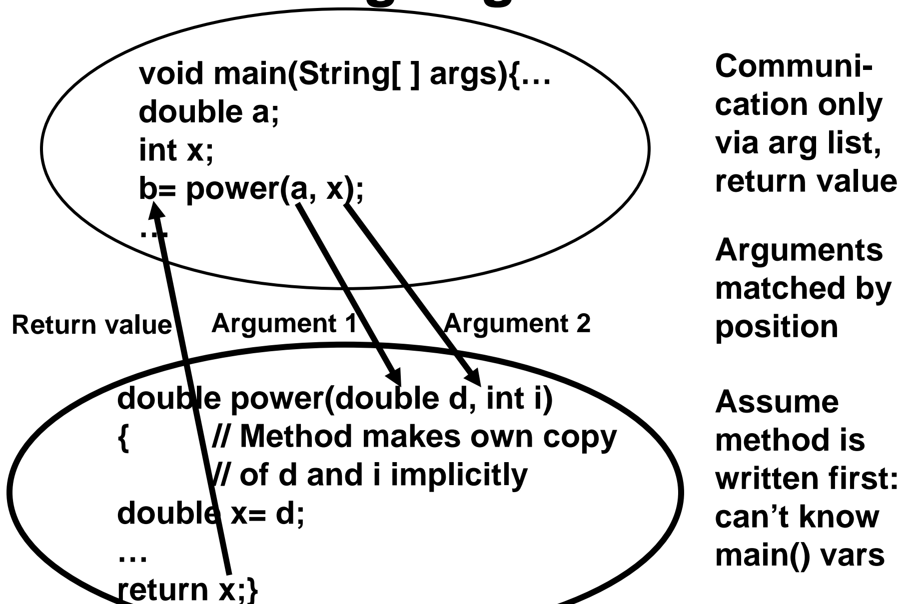
- Methods are the interface or communications between classes
  - They provide a way to invoke the same operation from many places in your program, avoiding code repetition
  - They hide implementation details from the caller or user of the method
  - Variables defined within a method are not visible to callers of the method; they have local scope within the method.
  - The method cannot see variables in its caller either. There is logical separation between the two, which avoids variable name conflicts.

Invoking a Method

```
public class Power1 {
    public static void main(String[] args) {
        double a= 10.0, b;
        int x= 3;
        System.out.println("Main: a= " + a + ", x= " + x);
        b= power(a, x);
        System.out.println("Main: Result b= " + b);
    }

    public static double power(double d, int i){
        System.out.println("Power: d= " + d + ", i= " + i);
        double x= d;            // Different x than main x
        for (int j= 1; j < i; j++)
            x *= d;             // x = x * d;
        return x;  }
}
```

Passing Arguments



Communication only via arg list, return value

Arguments matched by position

Assume method is written first: can't know main() vars

Call By Value

- **Java supports only call-by-value when passing arguments to a method:**
  - **The method makes a local copy of each argument**
  - **It then operates on that local copy**
  - **A method cannot change the value of an argument (variable) in the method that calls it**

Call By Value

- **A method can only send information back to its caller**
  - **Via the return value, and**
  - **By changing the data in objects passed to it (more on this later)**
- **The return value can be void, meaning nothing is returned. Method main() has a void return.**
- **The other mechanism, supported by C++ and other languages, is call-by-reference, which allows the method to change the arguments.**

Call By Value Example

```
public class CallByValue {
    public static void main(String[] args) {
        int i= 3;
        double d= 77.0;
        System.out.println("Main 1: i= " + i + ", d= " + d);
        triple(i, d);   // No return value
        System.out.println("Main 2: i= " + i + ", d= " + d);
        // Secondary part of example: argument conversion
        triple(i, i);   // Ok-Java converts int to double
        System.out.println("Main 3: i= " + i);
    }
    public static void triple(int ii, double dd) {
        System.out.println(“Triple 1: ii= " +ii+ ", dd= " +dd);
        ii *= 3;        // ii= ii*3;
        dd *= 3.0;
        System.out.println(“Triple 2: ii= " +ii+ ", dd= " +dd);
    }
}
```

Call By Value With Object

```
public class CallObjExample {
    public static void main(String[] args) {
        Demo d= new Demo(3);
        int i= 3;
        System.out.println("Main1: i= " + i + ", d.a= " + d.a);
        triple(i, d);   // No return value
        System.out.println("Main2: i= " + i + ", d.a= " + d.a);
    }
    public static void triple(int ii, Demo dd){
        System.out.println("T1: ii= "+ ii + ", dd.a= " + dd.a);
        ii *= 3;        // ii= ii*3;
        dd.a *= 3;
        System.out.println("T2: ii= "+ ii + ", dd.a= " + dd.a);
}  }
```

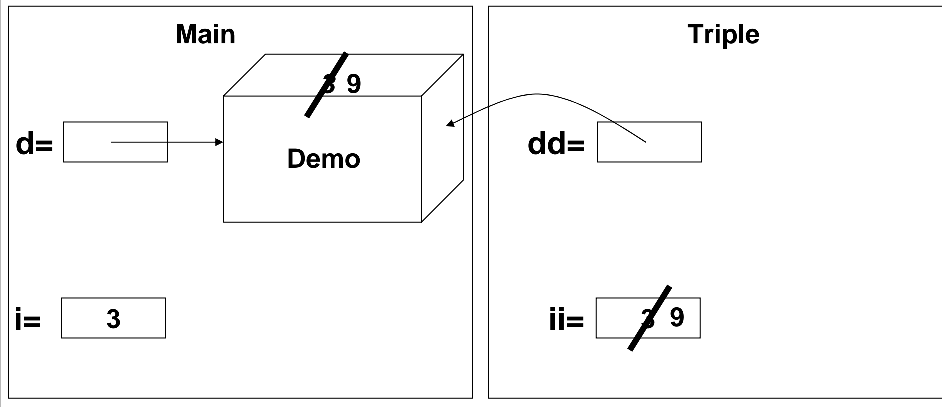
Call By Value With Object

```
class Demo {
    // public only to show call by value
    public int a;
    public Demo(int aa) {
        a= aa;
    }
}
```

Call by Value With Object

```
Demo d= new Demo(3);
int i= 3;
triple(i, d);
```

```
ii *= 3;
dd.a *=3;
```



Method Signature

- The name and list of arguments of a method is called its "signature"
- Within a class, methods can have same name as long as they have different signature
- Within a class, all methods with same name must have same return type.

Method Overloading

- Using same method name with different signatures is called "method overloading".
- Java selects which version of overloaded method to call based on number, type and order of arguments in the method's invocation
- Java will "promote" numerical data types "upwards" to match signatures

```
public class Overload {
  public static void main(String[] args) {
    System.out.println("First version="+ retVal("1.5E3") );
    System.out.println("Second version=" + retVal(1500) );
    System.out.println("Third version=" + retVal(1.5, 3.0) );
   }

    public static double retVal(String s) {
        return Double.parseDouble(s);}

    public static double retVal(int i) {
        return (double) i;  // cast optional here
    }

    public static double retVal(double m, double exp){
        return m * Math.pow(10.0, exp);
    }
}
```

String class

- **Part of Java system**
- **String class has special operator for concatenation**
- **String class allows constant initializer, as in**

`String testString = "Hello";`

String is not modifiable once created-"immutable"

Strings

```
public class StringExample {
    public static void main(String[] args) {
        String first= "George ";
        String middle= "H.W. ";
        String last= "Bush";
        String full= first + middle + last;
        System.out.println(full);

        // Testing for equality in strings (objects in general)
        String full2= "George H.W. Bush";
        if (full.equals(full2))     // Right way
            System.out.println("Strings equal");
        if (full == full2)          // Wrong way
            System.out.println("A miracle!");
        if (first == "George ")     // Wrong way, but works
            System.out.println("Not a miracle!");
```

Strings

```
// Modifying strings must be done indirectly
// Strings are constant
        middle = middle.substring(0, 2) + " ";
        full= first + middle + last;
        System.out.println(full);
        }
}
```

== vs equals()

```
public class EqualsTest {
   public static void main(String[] args) {
      DemoInt d= new DemoInt(3);
      DemoInt c = new DemoInt(3);
      DemoInt e = new DemoInt(4);
      if (d == c)  System.out.println("c == d");
          else System.out.println("c != d");
      if(c.equals(d)) System.out.println("c equals d");
          else System.out.println("c not equals d");
      if(c == e) System.out.println("c == e");
          else System.out.println("c != e");
      if(c.equals(e) System.out.println("c equals e");
          else System.out.println("c not equals e");
   }  }
```

== vs equals() – slide 2

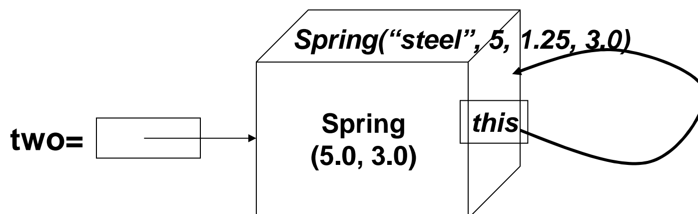
```
class DemoInt {
   public int a;       // Public only to show call by value
   public DemoInt(int aa) { a= aa; }

   public boolean equals(DemoInt q){
    if(a == q.a)
          return true;
    else
          return false;
   }
}
```

"this"- How an object refer to itself

- Objects are accessed by variables we called references.
- Suppose from code within an object, we want to refer back to the object.
- Example: Suppose we want to pass a reference to "ourself" as argument to a method of another object

```
public Spring(String m, double len, double md, double k) {
        material= m;
        length= len;
        maxDeflect= md;
        this.k= k;                    // "this"
}
public Spring(double len, double k) {
        this("steel", len, 0.25*len, k);     // "this"
}
```




```
two = new Spring(5.0, 3.0)
```

"this" is also used as shorthand to refer to other constructors for the class

this in a constructor

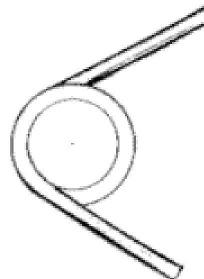
```
public class SimplePoint {
    private double x, y;                // Data members
    public SimplePoint() {              // Constructor
        x= 0.0;
        y= 0.0; }
     public SimplePoint(int x, int y) {
         this.x = x;
         this.y = y; }
    // Methods
    public double getX() { return x;}
    public double getY() { return y;}
    public void setX(double xval) { x= xval;}
    public void setY(double yval) { y= yval;}
    public void move(double deltaX, double deltaY) {
        x += deltaX;
        y += deltaY; }
}   // End of class SimplePoint
```

Springs

Compression

Tension

Torsion



F= k dx

Spring Class Constructors

```
class Spring {
    private String material= "steel";       // Initialized
    private double length;                  // Compress only
    private double maxDeflect;
    private double k;

    public Spring(String m, double len, double md, double k) {
        material= m;
        length= len;
        maxDeflect= md;
        this.k= k;                          // “this”
    }
    public Spring(double len, double k) {
        this("steel", len, 0.25*len, k);    // “this”
    }
    public Spring(double len) {
        this(len, 0.5*len);
    }
    public Spring() {
        this(10.0);    }
```

Spring Class Methods

```
    public String getMaterial() { return material; }
    public double getLength() { return length; }
    public double getMaxDef() { return maxDeflect; }
    public double getK() { return k; }    // Don’t need ‘this’

    public void setMaterial(String m) {material= m; }
    public void setLength(double len) {length= Math.abs(len);}
    public void setMaxDef(double m) {maxDeflect= Math.abs(m);}
    public void setK(double k) {this.k= k; }        // this

    public double getForce(double deflection) {
        if (deflect > maxDeflect)
            deflect= maxDeflect;
        return k*deflect;
    }
    public double getForce() {          // Overloaded methods
        return k*maxDeflect;
    }
}
```

Spring main()

```
public class SpringExample {
    public static void main(String[] args) {
        Spring one= new Spring("aluminum", 2.0, 1.0, 5.0);
        Spring two= new Spring(5.0, 3.0);
        Spring three= new Spring();     // 3 diff constructors

        double f1= one.getForce(0.5);
        double f2= two.getForce(1.5);
        double f3= three.getForce(0.1);
        System.out.println("f1: " + f1 + "\nf2: " + f2 +
              "\nf3: " + f3);

        double f4= one.getForce();      // Overloaded methods
        double f5= two.getForce();
        double f6= three.getForce();
        System.out.println("f4: " + f4 + "\nf5: " + f5 +
              "\nf6: " + f6);
        System.exit(0);
    }
}
```

Spring Class Design

- **All the Spring methods are public**
  - **Any method of any class can call these methods**
  - **Private methods can be used, as helpers or for tricky things that only the class itself should do**
- **Data fields in Spring are private**
  - **Only Spring methods can access them**
  - **Public data fields are almost never used**
- **Constructor name must be same as class name**
  - **Constructors are called with new only**
  - **Constructors cannot have a return value**

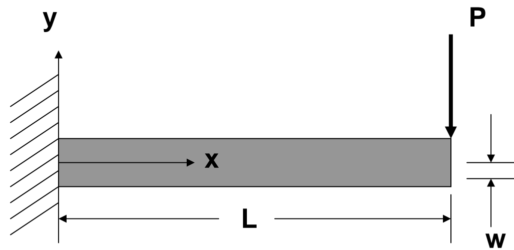
Spring Class Methods

- Why have all these methods?
  - Get methods are only access from other classes to Spring data
    - Allow us to reimplement Spring if we need
  - Set methods should do error checking
    - Our method should make sure that length>0, max deflection < length, etc.
- Overloaded methods must have different signatures (arguments)
  - Different return types cannot distinguish two otherwise identical methods
    - `int getForce(int a)` and
    - `double getForce(int b)` would not compile

Object Destruction

- Java reclaims object memory automatically using ‘garbage collection’ when there are no active references to the object
  - C++ requires the programmer to do this manually. You use ‘new’ sparingly in C++ because you have to use ‘delete’ when done, to avoid ‘memory leaks’
- Java has finalizers to clean up other resources (files, devices locked, etc.) when an object is destroyed
  - Informal advice: never use finalizers
  - They can invoke any object, so garbage collector can be wildly inefficient

Beam Exercise (optional if time allows)



Write a Java class to model this beam and compute its maximum deflection:

$w = - P L^3 / 3 E I$

where

- **P= load at end (1200 N)**
- **L= length of beam (20 m)**
- **E= elastic modulus (30000 N/ m^2)**
- **I= moment of inertia (1000 m^4)**
- **w= deflection (m)**

Beam Exercise, p.2

- **Data fields:**
  - Which are beam characteristics?
  - Which are external?
- **Write two constructors:**
  - One with all fields as arguments
    - Use same names for arguments and fields
  - One with only length argument
    - Other fields default as on previous slide
    - Use 'this' to invoke other constructor or rely on initialization
  - Use initialization in your class: assume you're dealing with many beams like the example beam
- **Write two methods to return the deflection w**
  - Use same method name for both (overloading)
  - One takes load as argument, 2nd takes load and units (ft or m) as a String; convert result using 1 m= 3.3 ft
- **Don't write any get or set methods**
- **Write a class with a main() to create a beam, compute its deflection and print the result**

Beam Exercise, p.3

- Optional, advanced:
  - Add dimensional analysis:
    - Store the units for each variable in the class
      - Decide how you'll code them (exponents, etc.)
    - Modify constructors to take unit arguments
    - Convert units if needed (N – lbf, m – ft)
      - 1 lbf= 4.4 N, 1 ft= 0.3 m
    - Make sure units match in the calculation
    - Output the units with the method result

Beam Class

```
class Beam {
    private double L;
    private double E= 30000.0;          // Initialization
    private double I= 1000.0;
    public Beam(double L, double E, double I) {
        this.L= L;          // this to access object fields
        this.E= E;
        this.I= I;
    }
    public Beam(double L) {
        this.L= L;          // Rely on initialization for others
    }                       // Could use this(L, 30000., 1000.);
    public double getDeflection(double P) {
        return -P*L*L*L/(3.0*E*I);
    }
    public double getDeflection(double P, String u) {
        if (u.equals("ft"))             // not ==
            return -3.3*P*L*L*L/(3.0*E*I);
        else
            return -P*L*L*L/(3.0*E*I);
    }  }
```

Beam Main()

```
public class BeamExample {
    public static void main(String[] args) {
        Beam one= new Beam(20.0);
        double w1= one.getDeflection(1200.0);
        System.out.println("w1: " + w1);
        double w2= one.getDeflection(1200.0, "ft");
        System.out.println("w2: " + w2);
        System.exit(0);
    }
}
```