Overloading

Overloading Methods

- a method that allows you to use one identifier to execute diverse tasks
- writing multiple methods in the same scope that have the same name but different parameter lists
- the parameter identifiers do not have to be different, but the parameter lists must satisfy one or both of these conditions:
  - must have different numbers of parameters
  - must have parameter data types in different orders

VALID = write <u>VALID</u> <u>AMBIGUOUS/NOT AMBIGUOUS</u> <u>FIRST/SECOND METHOD</u>

INVALID = write the reason

LAST TWO METHOD CALLS = display on the screen

```
public static void calculateInterest(double bal, double rate){
        double interest;
        interest = bal * rate;
        System.out.println("Simple interest on $" + bal + " at "
                + rate + "% rate is " + interest);}


public static void calculateInterest(double bal, int rate){
        double interest, rateAsPercent;
        rateAsPercent = rate / 100.0;
        interest = bal * rateAsPercent;
        System.out.println("Simple interest on $" +
                bal + " at " + rate + "% rate is " + interest);}
```

```java
public static void overload(int a)
public static void overload(int a, double b)
public static void overload(String a)
```

Constructor with Parameters

```
public class Employee{
  private int empNum;
  Employee(){
    empNum = 999;}
}
```

```
Employee partTimeWorker =
              new Employee();
```

```
public class Employee{
  private int empNum;
  Employee(int num){
     empNum = num;}
}
```

```
Employee partTimeWorker =
              new Employee(881);
```

Overloading constructors

```
public class Employee{
  private int empNum;

  Employee(int num){
    empNum = num;}

  Employee(){
    empNum = 999;}
  }
```

```
public class Student{
  private int stuNum;
  private double gpa;

  public Student (int stuNum, double gpa){
    stuNum = stuNum;
    gpa = gpa;}

  public void showStudent(){
    System.out.println("Student #" + stuNum + " gpa is " +
       gpa);}
}


 public class TestStudent{
   public static void main(String[] args) {
     Student aPsychMajor = new Student(111, 3.5);
     aPsychMajor.showStudent();}
 }
```

```
public class Student{
  private int stuNum;
  private double gpa;

  public Student (int stuNum, double gpa){
    this.stuNum = stuNum;
    this.gpa = gpa;}

  public void showStudent(){
    System.out.println("Student #" + stuNum + " gpa is " +
       gpa);}
}
```

```
 public class TestStudent{
   public static void main(String[] args) {
     Student aPsychMajor = new Student(111, 3.5);
     aPsychMajor.showStudent();}
 }
```

Static and final fields

- Fields declared to be static are not always final. Conversely, final fields are not always static. In summary:
  - If you want to create a field that all instantiations of the class can access, but the field value can change, then it is static but not final.
    - `public static int a;`
  - If you want each object created from a class to contain its own final value, you would declare the field to be final but not static.
    - `public final int b;`
  - If you want all objects to share a single nonchanging value, then the field is static and final.
    - `public static final int b=5;`