```
public class MagicConch
{
   private final String specialMessage;

   public MagicConch(String sm) // constructor (1)
   {
      specialMessage = sm;
   }

   public MagicConch() // constructor (2)
   {
      this("not 4 U");
   }
}
```

Continue on next page

In main:

```
   MagicConch m = new MagicConch("In your dreams");  // calls constructor that takes in one argument
   MagicConch p = new MagicConch(); // calls constructor that takes no arguments in
```

m and p will each have their own specialMessage data fields. This is because specialMessage was not declared static (i.e., it is non-static).

m's specialMessage is "In your dreams"
p's specialMessage is "not 4 U"

A static class variable is shared by all instances of the class.
Each instance of the class will have its own copy of the class' non-static variables.

```
public class NonMagicalConch
{
   private static int count; // Shared by each instance. Exists outside of the instances.
   private String boringMessage; // One of these exists in each class instance

   // ...
}
```

A static method does not have an invoking object.
The invoking object is the object that comes before the . in the method call (if the method is non-static).
Ex:

```
    Circle c = new Circle(4);  // Suppose we have a Circle class
    int r = c.getRadius();       // c in this method call is the invoking object
```

A non-static method has an invoking object.

Since a static method does not have an invoking object, you cannot use the keyword this in it.

What is wrong with adding the following methods to class NonMagicalConch?

```
    public String speech()   // This method is OK
    {
        return boringMessage + " Amen.";
    }

    public static String multiSpeech()
    {
        return speech() + " x " + count;  // Error: We cannot call speech() from a static context
    }                                     //          because speech() is non-static.
```

Answer:
In multiSpeech() method speech() cannot be called because speech() is shorthand for this.speech(), and becuse multiSpeech() is static it does not have an invoking object (cannot use keyword this in multiSpeech()).

It is a common design error for new programmers to declare a method as non-static whe it should be static.
A method should be static if it does not need an invoking object.

Example:

```
public class MyMath
{
   public int square(int x) // Design error, method square does not use an invoking object so it should be static
   {
       return x*x;
   }
}
```

Correction is to make method square static.

Suppose we have a class called Person, and method Person has a getPopulation() method that returns the total number of Person objects created in the program.

Q. Should getPopulation() be static or non-static?
A. Static, because population does not depend on a specific Person.

To call a static method (outside the class) you put the name of the class before the . in the method call instead of putting the invoking object there.

```
int pop = Person.getPopulation();
```

The class user is the portion of code ouside the class that uses the class (and it instances).
The class designer is the code for the class.

A class' public interface is its set of methods and data fields that are accessible by the class user (i.e., the class' set of public members). This is what the class user sees and uses to interact with the class and it instances.

A class' public interface should only consist of the members that the class user should have access to.
Do NOT make a member public if it should not be accessed by the class user.

Class members that are only to be used internally within a class shoud be declared private.

All the values of the non-static data fields of an object together form the object's state. The class user should not be able to put an object in an invalid state.

Encapsulation is
    the restriction of access to data fields, and
    the hiding of implementation details from the class user.

Encapsulation helps separate implementation details from an objects' behavior.
It is also used to prevent the class user from putting an object in an invalid state.

To restrict access to a data field we make it private.
To allow the class user to read the data field, we make a public get method for it.
To allow the class user to modify the data field, we make a public set method for it.

```
public class Color
{
   private int code;

   public int getCode() // the get method for code
   {
      return code;
   }

   public void setCode(int newCode) // set method for code. ONLY add if you want the
   {                                                      // class user to be able to change the value
       code = newCode;
   }
}
```

In class user code:

```
   Color c = new Color();
   int a = c.getCode();
   c.setCode(0xff0000);
```

Packages
In Java, a collection of classes can be grouped together into a package.

Examples: java.util is a package included with Java. Some classes in it are Scanner and Random.

To use class Random, we must import it at the top of our file:
```
import java.util.Random;
```

Alternatively, we can specify the full package name everywhere we referenc class Random:
```
public class A
{
    private java.util.Random r;
}
```

We can import all classes from java.util with:
```
import java.util.*;
```

Each package is structured as a hierarchy of folders matching the package name.

To add a class to a package:
- place its .java file in the folder for the package
- at the top of the .java file (before any imports) write the line
```
package packageName; // replace packageName with the actual package name
```

Example:
package chicken.soup;

Classes in the same package do not need to import each other.

If we do not specify the package at the top of the file, then the class is put in the default package.

A class member with no access modifier (no public, private, etc. before it) will have package access. This means it can only be accessed by classes within the same package.

The JVM will automatically import package java.lang
So, we do not need to import java.lang