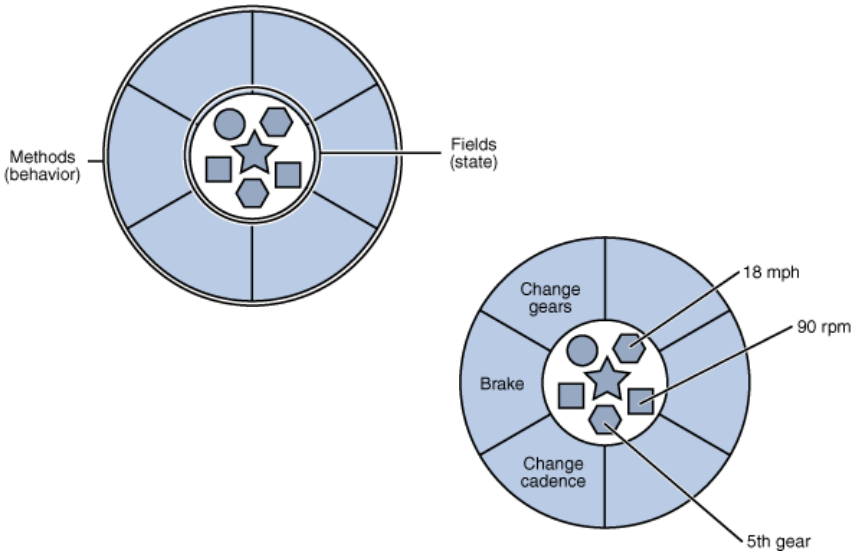
Classes and Objects

What are objects?

- Real-world objects share two characteristics: They all have *state* and *behavior*.
- Dogs have state (name, color, breed) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current speed) and behavior (changing gear, applying brakes).
- Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.



Software Objects

- Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* or *variables* and exposes its behavior through *methods* (functions in some programming languages).
- Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.
- Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming.

Advantages of using Objects?

- Modularity
- Information-hiding
- Code re-use
- Pluggability and debugging ease

Classes

- A *class* is the blueprint from which individual objects are created.
- Example
  - There may be thousands of other bicycles in existence, all of the same make and model.
  - Each bicycle was built from the same set of blueprints and therefore contains the same components.
  - In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles.

Defining a Class

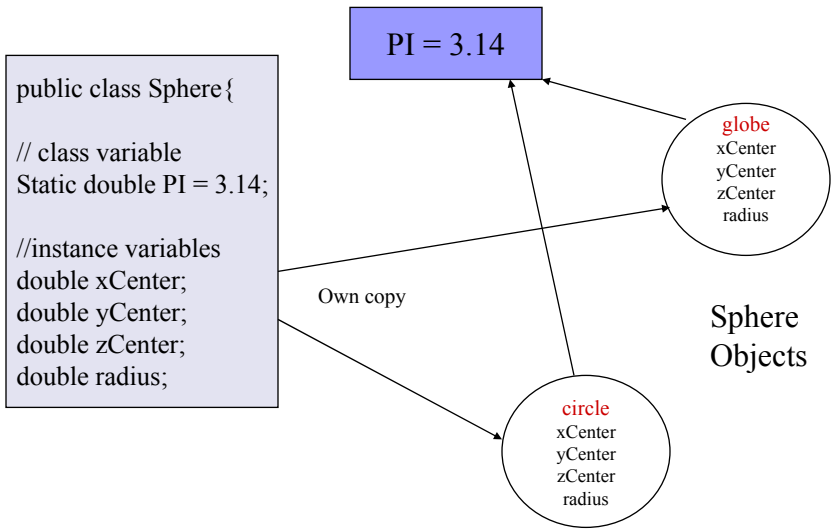
- **There are three kinds of things that you can include in a class definition**
  - Data members also called variables, fields.
  - Member methods
  - Initialization blocks
  - An instance of a class is technical term for an existing object of that class. So instantiation is making objects.
- **Data members**
  - These are the variables that store data items that typically differentiate one object of a class from another.
  - Variables in a class definition are not the same, there are two kinds
    - Instance variable
    - Class variables

Variables

- **Instance variable are those variables that are associated with each object uniquely.**
  - Each instance of the class will have its own copy of each of these variables**.**
  - Each object will have its own values for each instance variables that differentiate one object from the other of the same class type.
  - Declared in the usual way and can have an initial value specified.
- **Class variables are associated with the class and is shared by all objects of the class.**
  - There is only one copy of each of these variables no matter how many class objects are created.
  - They exist even if no objects of class have been created.
  - These variables are also called static fields because we use the keyword static when we declare them.

Variables

- **Why would you need two kinds of variables in a class definition?**
  - One use of class variables is to hold constant values that are common to all the objects of the class
  - Another use is to track data values that are common to all objects and that need to be available even when no objects have been defined. Like keeping count of no of objects created in the program



Methods

- **Member Methods**
  - These define the operations you can perform for the class, typically on the variables of the class
  - There are two kinds of methods
    - Instance methods
    - Class methods

Methods

- **Instance Methods**
  - These methods can only be executed in relation to a particular object
  - If no object exists, no instance method can be executed
  - Note: Although instance methods are specific to objects of a class, there is only one copy of an instance method in memory that is shared by all the objects of the class.
- **Class Methods**
  - You can execute class methods even when no objects of a class exist.
  - Like class variables, these are declared using the keyword static, so also called static methods
  - Static methods cannot refer to instance variables or call instance methods. Why?
  - Why main is always declared static?

Defining a Class

- **Defining Classes**
  - Use the keyword class followed by the name of the class followed by a pair of braces enclosing the details of the definition.
- **Member variables get initialized to default values if we don't initialize them explicitly**
  - Numeric fields initialized with zero, char fields with '\u0000', boolean with false and fields references are initialized with null

Defining Classes

- **Accessing Variables and Methods**
  - How can we access variables and methods defined within a class from outside it?
  - Accessibility is different for instance and static members
- **For Static members, we have got two choices**
  - Use class name followed by a dot operator followed by the static member name i.e Math.sqrt(Math.PI);
  - If reference to an object of the class type is available,then use reference name followed by dot operator, followed by the member name.
- **For Instance members**
  - They can be called only through the object reference

Defining Classes

- Class Declaration
  - class *MyClass* {
    //field, constructor, and method declarations
    }
- The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class:
  - constructors for initializing new objects,
  - declarations for the fields that provide the state of the class and its objects,
  - and methods to implement the behavior of the class and its objects.

Declaring Variables

- Variable declarations are composed of three components, in order:
  - Zero or more modifiers, such as public or private.
  - The variable's type.
  - The variable's name.

Declaring Methods

- Method declarations have six components, in order:
  - Modifiers—such as public, private, and others you will learn about later.
  - The return type—the data type of the value returned by the method, or void if the method does not return a value.
  - The method name
  - The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
  - An exception list—to be discussed later.
  - The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

```
return_type method_name(arg1, arg2, …. , argn){
        //code
}
```


Initialization Blocks

- **You can initialize data members with explicit values if you like, but in case initialization involves some calculations then you have to use initialization blocks**
- **An initialization block is a block of code between braces that is executed before an object of the class is created.**
- **There are two kinds of initialization blocks**
  - Static initialization block
  - Non-static initialization block

Initialization Blocks

- **Static initialization block**
  - Is a block defined using the keyword static
  - It is executed once when the class is loaded.
  - It can only initialize static data members of the class
- **Non-static initialization block can initialize both static and no-static data members**
  - Normally you don't initialize static data members using non-static initialization block
  - This block is executed each time an object is instantiated
- **You can have multiple initialization blocks in a class, in which case they execute in the sequence in which they appear.**

Static Initialization Block Example

```
class TryInitialization
{
  static int[] values = new int[10];               // Static array
member

  // Initialization block
//  static
  {
    System.out.println("Running initialization block.");
    for(int i=0; i<values.length; i++)
      values[i] = (int)(100.0*Math.random());
  }

  // List values in the array for an object
  void listValues()
  {
    System.out.println();                          // Start a new line
    for(int i=0; i<values.length; i++)
      System.out.print(" " + values[i]);           // Display values

    System.out.println();                          // Start a new line
  }
```

```
public static void main(String[] args)
   {
     TryInitialization example = new TryInitialization();
     System.out.println("\nFirst object:");
     example.listValues();

     example = new TryInitialization();
     System.out.println("\nSecond object:");
     example.listValues();
   }
}
```

Non-Static Init Block

```
class TryInitialization
{
 static int[] values = new int[10];              // Static array  member

 // Initialization block

 {
  System.out.println("Running initialization block.");
  for(int i=0; i<values.length; i++)
    values[i] = (int)(100.0*Math.random());
 }
```

Constructor

- When you create an object of a class, a special kind of method called a constructor is always invoked.
- If you don't define any constructor, the compiler will supply a default constructor in the class that does nothing
- **A constructor has two special characteristics which differentiate it from other class methods**
  - A constructor always has the same name as the class
  - A constructor never returns a value and you must not specify a return type even void is not allowed

Constructor

- **A constructor can have any number of parameters including none**
- **A constructor is called with *new* keyword when we make object of any class.**
- **If we define a constructor, then default constructor is not provided by the compiler**
- **You can have multiple constructors for your class**
- **Any initialization blocks that you have defined in a class are always executed before the constructor**

```
class Sphere
{
  static final double PI = 3.14; // Class variable that has a fixed valu
  static int count = 0;          // Class variable to count objects

  // Instance variables
  double radius;                 // Radius of a sphere

  double xCenter;                // 3D coordinates
  double yCenter;                // of the center
  double zCenter;                // of a sphere

  // Class constructor
  Sphere(double theRadius, double x, double y, double z)
  {
    radius = theRadius;         // Set the radius

    // Set the coordinates of the center
    xCenter = x;
    yCenter = y;
    zCenter = z;
    ++count;                    // Update object count
  }
```

```
// Static method to report the number of objects created
  static int getCount()
  {
    return count;               // Return current object count
  }

  // Instance method to calculate volume
  double volume()
  {
    return 4.0/3.0*PI*radius*radius*radius;
  }
}
```

Multiple Constructors

```
// Class constructor
  Sphere(double theRadius, double x, double y, double z)
  {
    radius = theRadius;          // Set the radius

    // Set the coordinates of the center
    xCenter = x;
    yCenter = y;
    zCenter = z;
    ++count;                     // Update object count
  }

  // Construct a unit sphere at a point
  Sphere(double x, double y, double z)
  {
    xCenter = x;
    yCenter = y;
    zCenter = z;
    radius = 1.0;
    ++count;                     // Update object count
  }
```


Creating Objects of a Class

- Declare the object
  - Sphere ball;
- Creating an Object
  - You must use keyword new followed by the constructor
  - ball = new Sphere(); or
  - ball = new Sphere (10,1.0,1.0,1.0);

```
public class CreateSpheres
{
  public static void main(String[] args)
  {
    System.out.println("Number of objects = " + Sphere.getCount());

    Sphere ball = new Sphere(4.0, 0.0, 0.0, 0.0);       // Create a
sphere
    System.out.println("Number of objects = " + ball.getCount());

    Sphere globe = new Sphere(12.0, 1.0, 1.0, 1.0);     // Create a
sphere
    System.out.println("Number of objects = " + Sphere.getCount());

    // Output the volume of each sphere
    System.out.println("ball volume = " + ball.volume());
    System.out.println("globe volume = " + globe.volume());
  }
}
```

Garbage Collection

- Objects are dynamically allocated
- How are these destroyed?
  - Answer: Garbage Collection
    - Automatic – occurs if objects exist and are no longer in use
    - If you create huge objects -> System.gc();

Method Overloading

- **Defining several methods in a class with the same name as long as each method has a set of parameters that is unique is called method overloading**
- **It is distinct from method overriding**
- **The signature – name of method, type and order of the parameters must be unique for each method**
- **Return type has no effect on the signature of a method**
- **How method overloading is useful?**
  - If methods do essentially the same things on different type of data
- **Constructors can also be overloaded (You can have multiple constructors for a class)**

this variable

- **Each object of the class have its own separate set of instance variables, but same instance methods**
- As there is only one copy of each instance method for a class, variable *this* allows the same method to work for different class objects
- **The variable *this***
  - Every instance method has a variable with the name ***this*** which refers to the current object for which the method is called
  - this is used implicitly by the compiler when your instance method refers to an instance variable of the class.

this variable

- Each time an instance method is called , the *this* variable is set to reference the particular class object to which it is being applied

Exercises

- 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?

What is the output?

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

Output

a.y = 5

b.y = 6

IdentifyMyParts.x = 2

b.x = 2

Exercise

- Create a class rectangle. The instance variables are length and width.
- Write a constructor to set the values for length and width to 1.
- Write a second constructor that takes 2 arguments length, width.
- Write get and set methods to get and set the values of instance variables.
- Write methods to calculate perimeter and area.
- Create two objects using two different constructors in the main method
- Print the perimeters and areas for both rectangles.

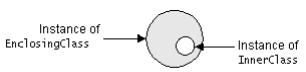
The finalize() method

- Sometimes an object will need to perform some action when it is destroyed
  - For example, when it is holding some non-java resource like a file or windows font
  - We want to make sure that they are released before an object is destroyed
- Finalization
  - We can define specific actions that occur when an object is just about to be destroyed
  - Done using the finalize() method
    - protected void finalize(){
      }
  - It is called by JVM before any object is destroyed.

Nested Classes

- **You can put the definition of one class inside the definition of another class. The inner class is called a nested class**
- **A top level class is a class that contains a nested class but is not itself a nested class**
- **A nested class can itself have another class nested inside it and so on**

```
public class Outside{
                    public class Inside {
                    //detail of nested class
                                        }
                    //more members of Outside class
                    }
```



Why Use Nested Classes?

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.

Nested Classes

- **The nested class is a member of the top level class**
- **The nested class can access attributes just like other class members**
- **When you declare an object of a class containing a nested class, no objects of the nested class are created unless the constructor of the enclosing class does so.**
  - Outside outer = new Outside( );
  - Outside.Inside inner = outer.new Inside ( );
- **You must refer to the nested class type using the name of the class as qualifier, but within non-static methods that are members of Outside, you can use the nested class name without any qualification**
  - Inside inner=new Inside( );

Nested Classes

- **Static methods cannot create objects of a non-static nested class**
- **To make objects of a nested class type independent of the objects of the enclosing class, what can we do?**
  - Declare the nested class as static
  - Outside.Inside inner=new Outside.Inside( );
- **A non-static inner class cannot have static members**
- **A non-static nested class can access any members of the top level class regardless of the access modifiers plus the Static members of any static nested class within the same top level class**