Class Structure
A class describes a set of objects

The objects are called instances of the class

A class describes:

- Fields (instance variables) that hold the data for each object
- Constructors that tell how to create a new object of this class
- Methods that describe the actions the object can perform

In addition, a class can have data and methods of its own (not part of the objects)

- For example, it can keep a count of the number of objects it has created
- Such data and methods are called static
- We are avoiding static data and methods for the time being
Defining a class

Here is the simplest syntax for defining a class:

```java
class ClassName {
    // the fields (variables) of the object
    // the constructors for the object
    // the methods of the object
}
```

- You can put `public`, `protected`, or `private` before the word `class`
- Things in a class can be in any order (I recommend the above order)
Defining fields

- An object’s data is stored in fields (also called instance variables)
- The fields describe the state of the object
- Fields are defined with ordinary variable declarations:
  - `String name;
  Double health;
  int age = 0;
- Instance variables are available throughout the entire class that declares them
Defining constructors

- A constructor is code to create an object
  - You *can* do other work in a constructor, but you *shouldn’t*
- The syntax for a constructor is:
  
  ```java
  ClassName(parameters) {
    ...code...
  }
  ``
  
  - The `ClassName` has to be the same as the class that the constructor occurs in
  - The `parameters` are a comma-separated list of variable *declarations*
public class Person {
    String name;
    int age;
    boolean male;
    Person (String aName, boolean isMale) {
        name = aName;
        male = isMale;
    }
}

Constructor

Parameters
Most constructors just set instance variables:

```java
public class Person {
    String name;
    boolean male;

    Person (String name, boolean male) {
        this.name = name;
        this.male = male;
    }
}
```
Defining a method

- A method has the syntax:

  ```
  return-type method-name(parameters) {
    method-variables
    code
  }
  ```

- Example:

  ```java
  boolean isAdult(int age) {
    int magicAge = 21;
    return age >= magicAge;
  }
  ```

- Example:

  ```java
  double average(int a, int b) {
    return (a + b) / 2.0;
  }
  ```
Methods may have local variables

- A method may have local (method) variables
- Formal parameters are a kind of local variable
  - int add(int m, int n) {
    int sum = m + n;
    return sum;
  }
- \(m\), \(n\), and \(sum\) are all local variables
  - The scope of \(m\), \(n\), and \(sum\) is the method
  - These variables can only be used in the method, nowhere else
  - The names can be re-used elsewhere, for other variables
Blocks (== Compound statements)

- Inside a method or constructor, whenever you use braces, you are creating a block, or compound statement:

```java
int absoluteValue(int n) {
    if (n < 0) {
        return -n;
    } else return n;
}
```
Declarations in a method

- The scope of formal parameters is the entire method.
- The scope of a variable in a block starts where you define it and extends to the end of the block.

```java
if (x > y) {
    int larger = x;
} else {
    int larger = y;
}
return larger;
```

*Illegal: not declared in current scope*
int fibonacci(int limit) {
    int first = 1;
    int second = 1;
    while (first < 1000) {
        System.out.print(first + " ");
        int next = first + second;
        first = second;
        second = next;
    }
    System.out.println();
}
The **for** loop

- The **for** loop is a special case
  - You can declare variables in the **for** statement
  - The scope of those variables is the entire **for** loop
  - This is true even if the loop is not a block

```java
void multiplicationTable() {
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++)
            System.out.print(" "+ i * j);
        System.out.println();
    }
}
```
Returning a result from a method

- If a method is to return a result, it must specify the type of the result:
  - `boolean isAdult ( ...`

- You must use a `return` statement to exit the method with a result of the correct type:
  - `return age >= magicAge;`
The keyword `void` is used to indicate that a method doesn’t return a value.

The `return` statement must not specify a value.

Example:

```java
void printAge(String name, int age) {
    System.out.println(name + " is " + age + " years old.");
    return;
}
```

There are two ways to return from a void method:

- Execute a return statement
- Reach the closing brace of the method
Sending messages to objects

- We don’t perform operations on objects, we “talk” to them
  - This is called sending a message to the object

- A message looks like this:
  
  \textit{object.method(extra information)}
  
  - The \textit{object} is the thing we are talking to
  - The \textit{method} is a name of the action we want the object to take
  - The \textit{extra information} is anything required by the method in order to do its job

- Examples:
  
  \texttt{g.setColor(Color.pink);}
  
  \texttt{amountOfRed = Color.pink.getRed( );}
Putting it all together

class Person {

    // fields
    String name;
    int age;

    // constructor
    Person(String name) {
        this.name = name;
        age = 0;
    }

    // methods
    String getName() {
        return name;
    }

    void birthday() {
        age = age + 1;
        System.out.println("Happy birthday!");
    }

}
Using our new class

Person john;
john = new Person("John Smith");

System.out.print (john.getName());
System.out.println(" is having a birthday!");
john.birthday();

- Of course, this code must also be inside a class!
A program consists of one or more classes.

Typically, each class is in a separate `.java` file.
null

- If you declare a variable to have a given object type, for example,
  - `Person john;`
  - `String name;`
- ...and if you have not yet assigned a value to it, for example, with
  - `john = new Person();`
    - `String name = "John Smith";`
- ...then the value of the variable is `null`
- `null` is a legal value, but there isn’t much you can do with it
  - It’s an error to refer to its fields, because it has none
  - It’s an error to send a message to it, because it has no methods
  - The error you will see is `NullPointerException`
Java has two kinds of methods: **static** methods and non-static methods (called **instance** methods)

However, before we can talk about what it means to be static, we have to learn a lot more about classes and objects

Most methods you write *should not*, and *will not* be static

Every Java program has a

```java
public static void main(String[] args)
```

method

This starts us in a “static context”

To “escape from static”, I recommend starting every program in a certain way, as shown on the next slide
Escaping from static

- class MyClass {

  public static void main(String[] args) {
    new MyClass().run();
  }

  void run() {
    // Your real code begins here
  }

}  

- You can replace the names MyClass and run with names of your choice, but notice that each name occurs in two places, and they have to match up
The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

— Joe Armstrong

Though this be madness, yet there is method in it.

— Shakespeare, *Hamlet*