# Classes: School is in Session

Aneesh Karve

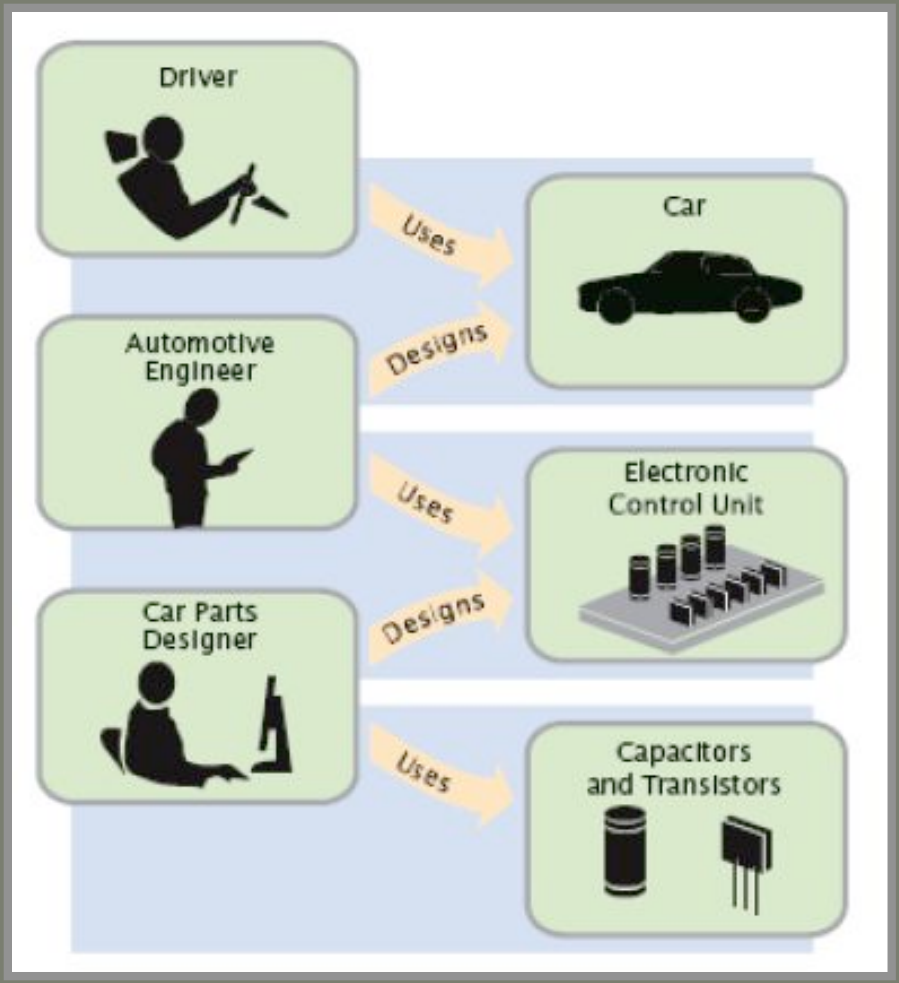
# Abstraction

- simplified, *high level* view
- captures essential properties

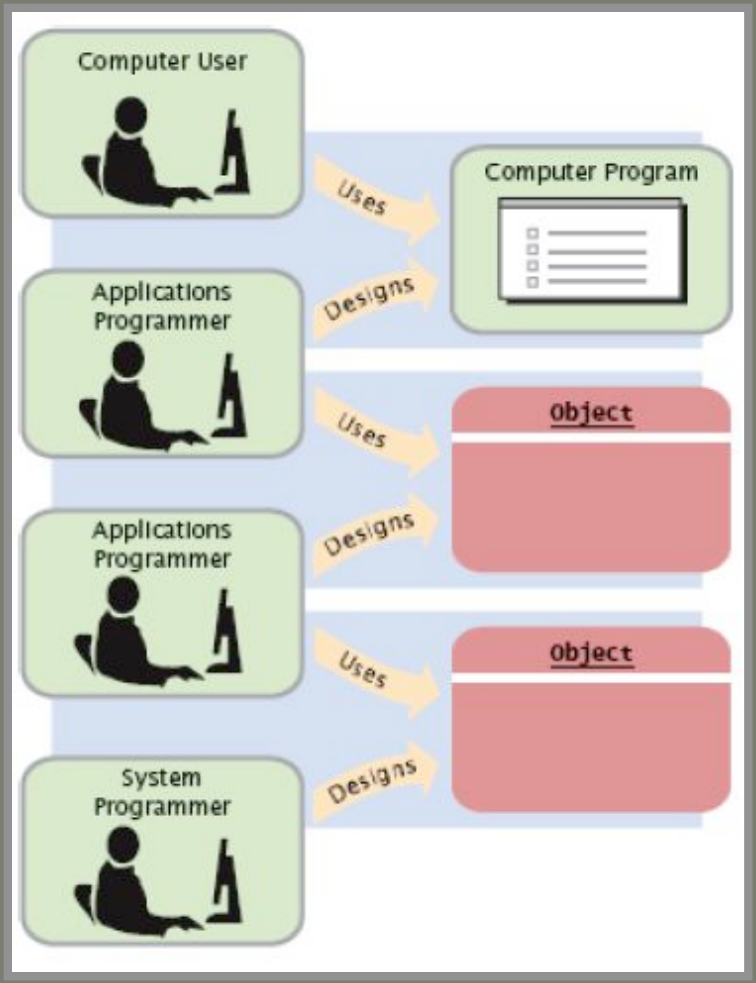
# Encapsulation

- public interface encapsulates (hides) the private implementation
- allows non-experts to use expert implementations as a “black box”

# Abstraction in automobiles



# Abstraction in software



# Object-oriented Programming (OOP)

- way of thinking about complex problems
- provides abstraction and therefore power
- build complex programs out of black box components

# Advantages of OOP

- makes programs easier to think about
  - fewer bugs
  - easier to maintain
- can create custom classes (types)

# Designing Classes

- first think of behavior (public interface); worry about implementation details later

# Access modifier

- keyword which determines availability of class, field, or method
- examples
  - **`public`**
  - **`private`**

# `private` enforces encapsulation

```
public class Capsule{
    private String secret = "rosebud";

    pubilc void changeSecret(String code){
        secret = code;
    }
}
```

OK to access private fields from within class

```
public class Spy{
    public static void main(String[] args){
        Capsule c = new Capsule();
        String stolen = c.secret;
    }
}
```

Can't access from outside

# Class syntax

access modifier

class name

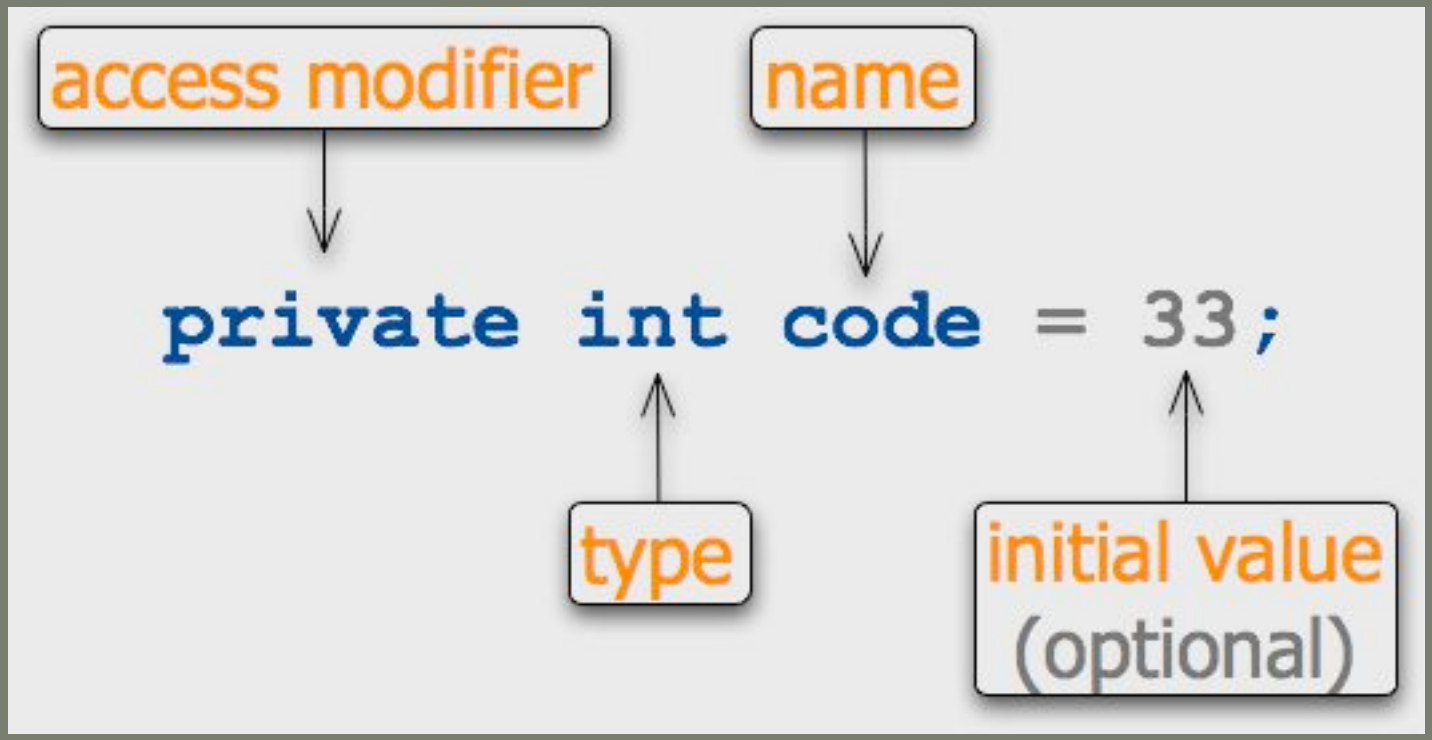
```
public class RockBand{
    fields (also called "data members" or "instance variables")
    private String name = "Sound Garden";
    ...
    constructors
    public RockBand(String name){...}
    ...
    methods
    public String getName(){...}
    ...
}
```

# Fields

- each object has the instance fields specified by its class
- generally, fields should be private

# Field declaration

- inside of class definition



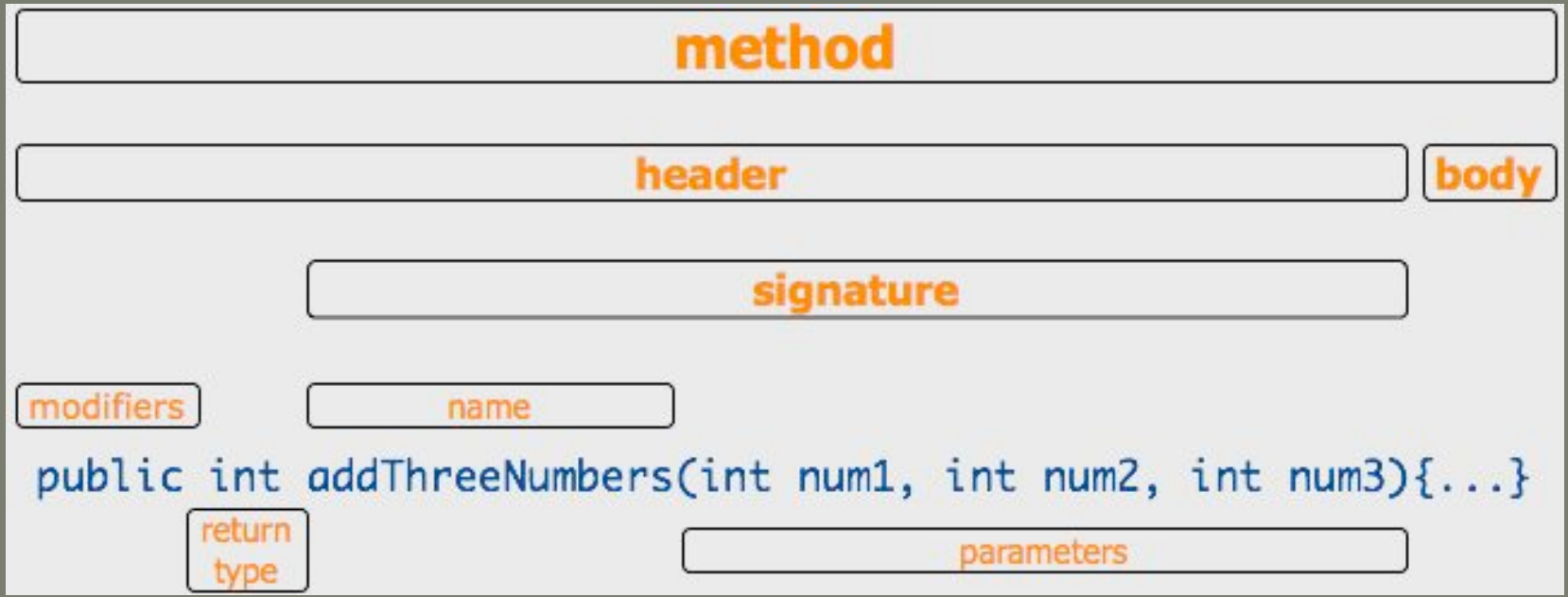
# Constructors

- create new objects
- initialize instance fields
- have same name as class

# Constructor example

```
public class IPod{
  //field
  private double cost;
  //constructor
  public IPod(double price){
     //initialize cost field
     cost = price;
  }
}
```

# Method anatomy



# Method Body

- where the work gets done
- consists of *statements*
- contains zero or more return statements

## `return`

- halts method execution
- may return a value

# Return type

- type returned by method

```
public int someMethod(){
    int x = doABunchOfCalculations();
    return x;
}
```

```
public void anotherMethod(){
    //code goes here
    return;
}
```

methods that don't return a value have **return type void**

# this

- object reference to implicit parameter

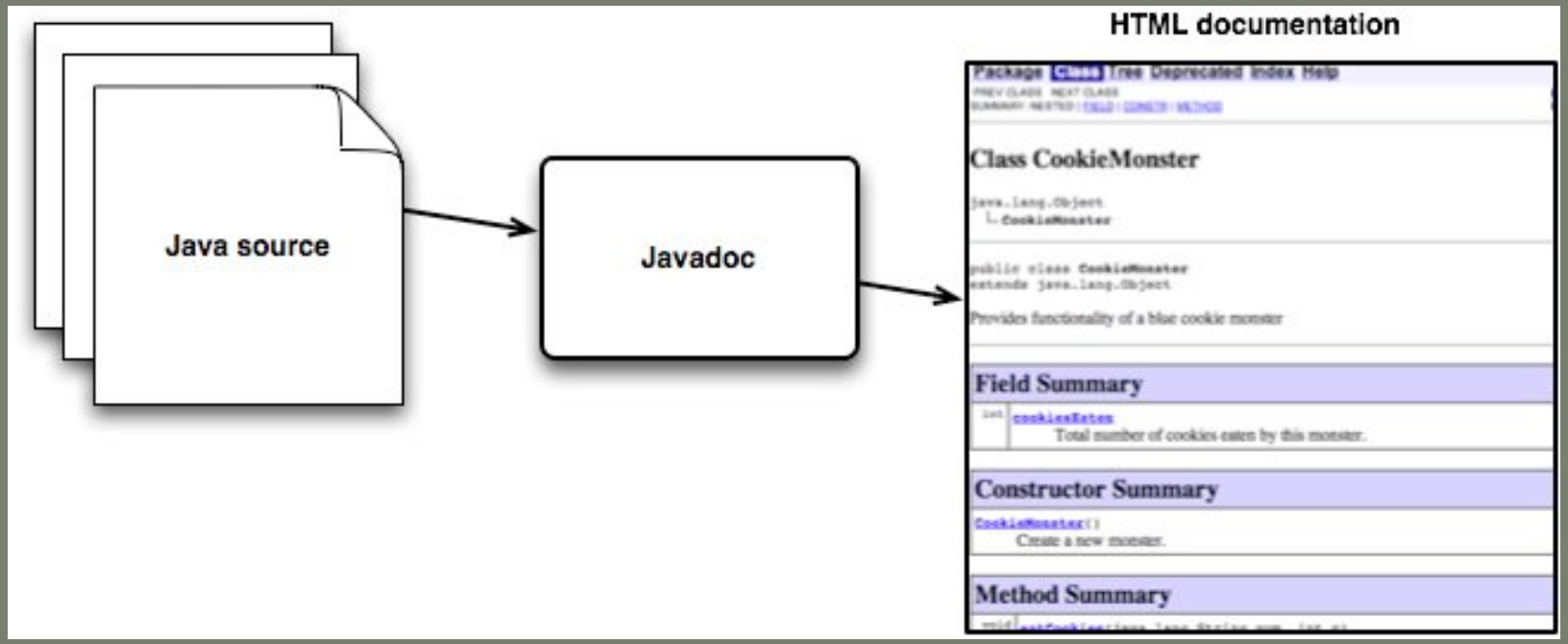
```
public class IPod{
    //field
    private double cost;
    //constructors ...
    //method
    public setCost(double cost){
        this.cost = cost;
    }
}
```

references field,
not parameter

# Javadoc

- tool for generating HTML documentation from source code
- use liberally (before each field, method, class, constructor)
- documentation is essential to intelligible, reusable code

# Javdoc in action



# Javadoc syntax

Javadoc comments start with two "*"s

Summary precedes first "."

you can format text (bold, italic, etc.)

Tags

```
/** This method is rad. Here's why:
 * <B>cookie monster is the man!</B>.
 * I think he should have his own show.
 * @param yum the cookie to eat
 * @param c number of yums to eat
 * @return number of crumbs flying from mouth
 */
public int eatCookies(String yum, int c){...}
```

# Categories of Variables

| | where declared? | where initialized? | lifetime? |
|---|---|---|---|
| • instance | class | constructor | object |
| • parameter | method prototype | method call | method execution |
| • local | block* | block (or else error) | block |

* a block of code is enclosed by brackets: {}

# Local vs. Parameter variables

- similarity: defined and initialized in methods
- difference: how initialized
  - parameter variables are initialized by calling the method
  - local variables initialized using assignment operator in method body

# Garbage collector

- reclaims memory from unreachable objects
- garbage not always collected immediately