Outline

- Class and method definitions
- Instance variables, constructors, methods
  - Remember from CH1: How is a class described? By attributes and behaviors!
- Access control
- Returning values from methods
- Variables and variable lifetime
- Object lifetime
class Laugher {

public Laugher() {
}

public void laugh() {
    System.out.println("haha");
}

}

Constructor function name must match the class name.
public void laugh()
{
    System.out.println("haha");
}

{}  Braces are delimiters that mark the start and end of the function body.
Laugher laugher = new Laugher();
laugher. laugh();

Sending message initiates execution of statements in method

Remember this?

public void laugh() {
    System.out.println("haha");
}

Constructor functions are automatically (implicitly) executed when a new object of the class is instantiated.
Another Method

If we want to change output when we send the message

Pass parameter to method - used as variable in program

Overloading

Constructor functions can be overloaded the same as any other function.

```java
public void laugh(String syllable) {
    System.out.print(syllable);
    System.out.println(syllable);
}
```

```java
laugher . laugh(”yuk”);
```
class Laugher {

    public Laugher() {
    }

    public void laugh() { // “default” function
        System.out.println(”haha”);
    }

    public void laugh(String syllable) { 
        System.out.print(syllable);
        System.out.println(syllable);
    }

} 

How could this class variation be modified to always display “laughing: ” before it displays anything else?

Constructor functions have no return type because they are used for object initialization.
Class Outline

- Class template organization:

```java
class Classname {
    // one or more constructor definitions
    public Classname () {
    }

    // zero or more function definitions
    public returnType FunctionName(parameters) {
        // function body statements
    }

    // zero or more function definitions
    public returnType FunctionName(parameters) {
        // function body statements
    }
}
```
Want to be able to change default syllable

**Default** constructor sets *default* syllable

```java
Laugher laugh1 = new Laugher();
laugh1.laugh(); //prints “haha”
```

Constructor with *argument* allows change

```java
Laugher laugh2 = new Laugher(“hee”);
launder2.laugh(); //prints “heehee”
```

To do this we need to understand *instance variables*
**Instance Variables**

- Variable stored as data belonging to object
- Called an *instance variable*
  - *Each object of the class has its own specific copy of the instance variables*
- *Declare* the variable in the class definition
  ```java
  private String defSyllable;
  ```
  …but do not create the *object* in the same place!

- Can be used in any method of class (scope)
- Should be initialized in the constructor
class Laugher {

    public Laugher() {
        defSyllable = "ha";
    }

    public Laugher(String syl) {
        defSyllable = syl;
    }

    // other methods here

    private String defSyllable; // instance variable
}

Constructors used to **initialize** instance variables

Constructors used to initialize instance variables.
class Laugher {

// constructors defined here

public void laugh() {
    System.out.print(defSyllable);
    System.out.println(defSyllable);
}

// other methods here

private String defSyllable;
}
class LaughABit {
    public static void main(String[] arg) {
        Laugher l1 = new Laugher("har");
        l1.laugh();
        Laugher l2 = new Laugher("yuk");
        l2.laugh();
    }
}

Once set by the constructor for an object the default syllable cannot be changed, (by the current class implementation). How could this behavior be altered to allow for the default syllable to be changed later?
Class definition includes information about access to methods and instance variables of class

- **public** – indicates that the method or variable can be used outside the class
- **private** – indicates that the method or variable can only be used inside the class

General rule:
- methods are public, unless they are “helpers” for other methods
- variables are private
A class definition must be stored in a .java file with the same name.

Ex. the class `Laugher` must be in `Laugher.java`.

Two classes may **not** have the same name.
```java
class Recording {
    public Recording(String t, String a) {
        title = t;
        artist = a;
    }
    public String getTitle() {
        return title;
    }
    private String title;
    private String artist;
}
```
Methods that return a reference value must use the `return` statement.
Recording rec;
rec = new Recording("Crossroads","Robert Johnson");

String artist;
artist = rec.getArtist();

It is valid to define variables with the same name as class instance identifiers outside the class. However, they are treated as different variables.

public String getArtist() {
    return artist;
}

The returned reference value replaces the message in the statement.
Kinds of Variables

- **(Formal) Parameters**
  - Arguments in method definition
  - Used to receive arguments in message

- **Local variables**
  - Declared within a method
  - Used to save computations local to method

- **Instance variables**
  - Declared in class definition
  - Used to store attributes of object
class Recording {

    public Recording(String t, String a) {
        title = t;  // t is a parameter
        artist = a; // a is a parameter
    }

    public String getTitle() {
        return title;
    }

    public String getArtistandTitle() {
        String temp = artist.concat(“-”).concat(title);
        return temp; // temp is a local variable
    }

    private String title;    // title is an instance var
    private String artist;   // artist is an instance var
}
public void generate(String s, PrintStream p) {
    // implementation of function goes here
}

generate("message", System.out);  // OK
generate("message");            // No good
generate(System.out, "message"); // No good
generate("message", System.out, "blah");  // No good

Number and type of arguments **must match** prototype. Parameters only exist while the method is executing! Parameters can only be accessed in their own method!
public String translate(String s) {
    String result;  // local variable
    result = (s.substring(1,5)).concat(s.substring(0,1));
    return result;
}

Local Variables only exist while the method is executing!
Local Variables can only be accessed in their own method!
Lifetime of Variables

- *Lifetime* – when variable is accessible

- It depends on the variable…
  - *Parameter and local variables* – only exist within each execution of method
  - *Instance variables* – exist as long as object does
Accessibility of Variables

- **Parameter and local variables**
  - only accessible within method – during lifetime

- **Instances variables** accessible during lifetime:
  - only accessible to methods of the class itself if the variable is declared **private**
  - to **any** method if the variable is declared **public**
Visibility of Variables

```java
class Example {
    public void doSomething(String s) {
        String name;
        s = "value one";
        name = "value two";
    }
    public void doSomethingElse() {
        s = "value three";
        name = "value four";
    }
    private String s;
    private String name;
}
```

A local declaration hides an instance variable
Lifetime of Objects

- An object exists from creation until it has no more reference variables to it
- Generally an object only has one reference variable that refers to it
  - Sometimes you don’t directly maintain the reference yourself… for example: InputStreamReader
- When lifetime of that one variable ends, the object is destroyed
**Example of Object Lifetime**

```java
void doSomething() {
    String s = new String("a story that is true");
    System.out.print(s);
}
```

Object creation

Object destruction

Say that object is **destroyed** when the variable `s` “goes out of scope” – the lifetime of `s` ends
public static void main() {
    Laugher l = new Laugher(new String("ha"));
}

class Laugher {
    ...
    public Laugher(String syl) {
        defSyllable = syl;
    }
    ...
    private String defSyllable;
}

What is lifetime of string created when constructing the Laugher object l?
A last word about **return** values

- We’ve seen it with strings…
  ```java
  String getTitle(){
    return title;
  }
  ```

- Can also be other types:
  - An **integer**:
  - A **boolean** value:
  - Even a class you define
    (You could even return a “Recording”!)

---

06. User Classes
A last word about **return** values

- **An integer:**
  ```java
  String myString = "Pete";
  myString.length(); // returns 4
  ```

- A boolean value (true or false):
  ```java
  if (myString.equals("Pete") )
      anotherObject.doSomething();
  ```
Designing Classes: Phase 1

- Methodical Approach To Designing Classes
- PHASE 1: SPECIFYING THE CLASS
  - Decide on the BEHAVIOR the class provides
    - (decide what METHODS to provide)
  - Determine the way the class will be used: INTERFACE
    - (choose the PROTOTYPES of the methods)
  - Write a SAMPLE program using the class as designed:
    - (CHECK the DESIGN)
  - Write a SKELETON of the class definition:
    - (just the PROTOTYPES with EMPTY METHOD BODIES)
PHASE 2: IMPLEMENTING THE CLASS

- Write the method bodies and declare any instance variables needed.

GUIDELINES:

- Start with the method that looks easiest.
- Feel free to switch to another method before finishing the one you are working. (You can always come back.)
- Focus on methods, not instance variables.
- When a method needs a reference to another object, it will get it from either a parameter or an instance variable.
- If no parameter, then an instance variable is needed.
- Instance variables must be initialized before used: often by constructor.