

Classes and Objects: Members, Visibility

Lecture 2

Members

- Two kinds of members in a class declaration
 - Fields:** data (determines the *state*)
`String color;`
`int length;`
 - Methods:** functions (access and/or modify the state)
`int sharpen (int amount) {`
 `length = length - amount;`
 `return length;`
`}`
- (Much later: nested classes and nested interfaces)

Visibility

- Members can be private or public
 - member-by-member declaration
`private String color;`
`public int length;`
`public int sharpen (int amount) { . . . }`
- Private members
 - Can be accessed only by instances of same class
 - Provide concrete implementation / representation
- Public members
 - Can be accessed by any object
 - Provide abstract view (client-side)

Example

```
class PencilA {  
    private String color;  
    private int length;  
    private boolean isValid(String c) {...}  
    public PencilA(String c, int l) {...}  
    public String toString() {...}  
    public void setColor(String c) {...}  
}  
  
class CreatePencilA {  
    public void m() {  
        PencilA p = newPencilA("red", 14);  
        p.setColor("blue");  
        p.color = "blue";  
    }  
}
```

OK → p.setColor("blue"); ← Compile-time Error

Example

- See PencilA.java
 - Concrete state (ie representation) is hidden from clients
 - Abstract state (ie client-side view) is accessed and manipulated through public methods
- See PencilB.java
 - Exact same behavior as far as the outside world is concerned

Best Practices: Member Declarations

- Group member declarations by visibility
 - Java's convention: private members at top
- No fields should be public
 - Common (bad) idiom: Public "accessor" methods for getting and setting private fields
`class Pencil {`
 `private int length;`
 `public int getLength() { . . . }`
 `public void setLength() { . . . }`
`}`
 - Better idiom: Public members allowing observing and controlling *abstract state* only (client view)

Method Invocation

- Syntax: *objectreference.member*

```
p.color = "red";
p.toString().length();
```
- Reference is implicit inside same object

```
class Pencil {
  private String color;
  public Pencil() {
    color = "red";
  }
}
```
- Explicit reference to same object available as **this** keyword (from within the object itself)

```
this.color = "red";
```

Best Practices: Formal Arguments

- Constructor arguments that are used directly to set object fields can be given the same name as the field
 - Formal argument "hides" class field variable
 - Refer to class field variable using explicit **this**

```
class Pencil {
  private int length;
  Pencil(int length) {
    this.length = length;
  }
}
```

Parameter Passing

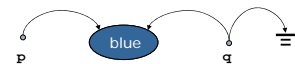
- Arguments are pass-by-value (copy)

```
public void modify (int a) {
  a = 6;
}

public void simple () {
  int b = 3;
  modify(b);
  // now b == ? still 3! ie unchanged
}
```
- When the parameter is an object reference, it is the *reference* being copied *not* the object itself!
 - As with =, this creates an alias

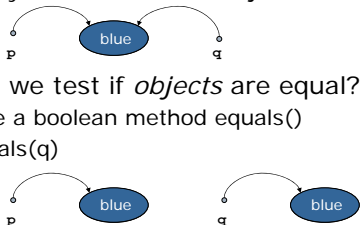
Alias by Parameter Passing Example

- ```
public void modify(Pencil q) {
 q.setColor("blue");
 q = null;
}

public void simple () {
 Pencil p = new Pencil("black");
 modify(p);
 //color of p is ? "blue"! ie changed
}
```
- 

## Testing for Equality

- For references p, q consider: p == q
  - Compares *references* for equality
  - Do they refer to the same object?
- How do we test if *objects* are equal?
  - Define a boolean method equals()
  - p.equals(q)



## Method Overloading

- A class can have more than one method with the same name as long as they have different *parameter lists*  

```
class Pencil {
 ...
 public void setPrice(float newPrice) {
 price = newPrice;
 }
 public void setPrice(Pencil p) {
 price = p.getPrice();
 }
}
```
- How does the compiler know which method is being invoked?
  - Answer: it compares the number and type of the parameters and uses the matched one  

```
p.setPrice(3.4);
```
- Differing *only* in return type is not allowed

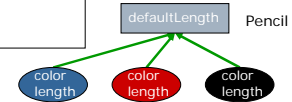
## Multiple Constructors

- Default constructor: no arguments
    - Fields initialized explicitly in declaration or implicitly to language-defined initial values
    - Provided automatically only if no constructor defined explicitly
- ```
class Pencil {
    String color; //initialized implicitly to null
    int length = 14; //initialized explicitly
}
...
```
- Copy constructor: one same-class argument
 - One constructor can call another with `this()`
 - If another constructor called, must be the first statement
- ```
Pencil (Pencil p) {
 this(p.color); //must be 1st line
 length = 10;
}
```

## Object vs Class Members

- Class member: only one copy, which is *shared* by all instances
  - keyword: `static`

```
class Pencil {
 private static int defaultLength;
 private String color;
 private int length;
 . . .
}
```



## Instance vs Static Members

- Static members available even before instances (objects) are created!
    - syntax (outside class): `classname.member`  
`Pencil.defaultLength++;`
    - inside class: `classname` is optional
  - Conversely, static members can not access instance members
    - `this` reference can not be used
- ```
public static void reset () {
    length = defaultLength;
}
```
- Compile-time Error

Best Practices: Static Members

- Do *not* access static members through object references
 - Use class names instead
 - Do this: `t = Pencil.defaultLength;`
 - Not this: `t = pl.defaultLength;`
 - This applies within a class too
- ```
class Pencil {
 private static int defaultLength;
 public void reset() {
 length = defaultLength; //correct
 length = Pencil.defaultLength; //better
 }
}
```

## Example: `println`

- `System.out.println("Hello");`
- What is *System*?
  - a class from Java library
  - see API documentation: `java.lang.System`
- What is *out*?
  - Static member of `System` (available from class)
  - type `PrintStream`
- What is `println`?
  - overloaded method in `PrintStream`
  - different versions for printing string, int, boolean...

## Example: `main()`

```
class HelloWorldApp {
 public static void main(String[] args) {
 . . .
 }
}
```

- `public`: so that JVM can run this method
- `static`: no instances of class created (yet)
- `void main(String[])`: required signature
  - JVM looks to invoke the method with this name
- `args`: array of command-line arguments
  - Any name can be used for formal parameter
  - "args" is just Java convention

## Miscellaneous Language Features

Computer Science and Engineering @ The Ohio State University

### □ Array length

- Set (at run time) and can not change  
`int[] ids = new int[rosterSize];`
- Available as a property with `.length`  
`void examine (int[] ids) {`  
    `for (int i = 0; i < ids.length; i++) {...}`

### □ Primitive types can be *cast*

- Widening is automatic (ie implicit)  
`long j = 12; //int to long (widening)`  
`long k = i; //int to long (widening)`
- Narrowing requires explicit cast  
`int i = 12L; //error: requires cast`  
`int i = (int) 12L; //long to int (narrowing)`  
`byte j = (byte) i; //int to byte (narrowing)`