What We Know

- Build programs with objects and interactions (or messages)
- Classes allow us to create objects
- Methods allow messages and provide object behavior
To design a class:

- Decide on *object behavior*
  - What (not how) a class functions
- Determine the *interface* of the class
  - means determining *prototypes* of methods
- Write a *sample program* using the class
  - allows you to ask/see if the design makes sense
- Write a *skeleton* of the class
  - class definition with prototypes
  - *empty method bodies!*
- Implement the class method *Java code*
  - Begin with whichever method seems logical to start
  - Decide on instance, and local variables while coding as needed.
Motivation:
- Let’s say we want to be able to do interactive input/output without having to worry about details of *how* that is done

Desired behavior:
- *Prompt for* and *read* String input from keyboard
- Write messages to console *w/o buffering delays*
- Create object without explicit reference to System classes that are needed
InteractiveIO Interface

- Creating objects
  ```java
  InteractiveIO interIO;
  interIO = new InteractiveIO();
  ```

- Writing messages
  ```java
  interIO.write("Your score is 12 out of 100");
  ```

- Prompting
  ```java
  String s;
  s = interIO.promptAndRead("What is your name?");
  ```
Prototypes for Interface

- Constructor – want default
  ```java
  public InteractiveIO()
  ```

- Write method
  ```java
  public void write(String s)
  ```

- Prompt-and-read method
  ```java
  public String promptAndRead(String s)
  ```

Focus design on parameters and return values.
// a paradigmatic example of using our class

import java.io.*;

class TryInteractiveIO {
    public static void main (String[] arg) throws Exception {
        InteractiveIO interIO;
        String name;
        interIO = new InteractiveIO; //create object
        name = interIO.promptAndRead("Gimme your name: ");
        interIO.write(name);
    }
}

class InteractiveIO {
    // creates object capable of performing I/O
    public InteractiveIO () {
        // statements will go here
    }
    // Writes s to the console
    public void write(String s) {
        // statements will go here
    }
    // Writes s to console
    // reads and returns string from keyboard
    public String promptAndRead (String s) {
        // statements will go here
    }
}

Decisions on parameter and return value types must be made here.
Implementing a Class

Choose:
- What instance variables are there?
- What statements implement the behavior of each method?

There are many ways to implement a class
All implementations must have the same interface
Process of Implementation

- You may work on any method at any time

- Decisions you make about using instance variables for one method may affect others

- One strategy: work on methods first
  - decide what instance variables are needed
  - then add variables and constructors
Writing ‘Write’ Method

- Behavior: immediately output text
- Can be done with
  ```java
  System.out.print(s);
  System.out.flush();
  ```
- Does not require instance variables

If the implementation of a method is too complex, then break it up into multiple helper methods.
Write Method

// Writes string s to console
public void write(String s) {
    System.out.print(s);
    System.out.flush();
}

Writing ‘promptAndRead’

- **Behavior:**
  - print prompt, (first)
  - read response, (second)
  - return response (third)

  Outline the steps necessary to implement the methods. Do not begin by trying to write code.

- **First** is same as write method! *(REUSE!)*

- **Second** requires BufferedReader object
  - How will we set up the BufferedReader?
  - Local variable? Parameter? Instance variable?

- **Third** is return statement

  Break outline steps up into simpler steps until each can be implemented as a single Java statement.
Outline of ‘promptAndRead’

When writing method identify steps to get behavior

```java
public String promptAndRead(String s) {
    // print prompt
    // read response
    // return response
}
```
Programming in Java

Returning Value

```java
public String promptAndRead(String s) {
    String response;
    // print prompt
    // read response
    return response;
}
```

Need local String variable to store response

Need return statement
Choices for Printing Prompt

Which should we do?

1. Use two lines in write method
2. Use the write method
   - We already wrote it!
3. Use a “helper” method for this
   and the write method

When more than one method is doing the same thing, the common code should be moved to a private helper method.
public String promptAndRead(String s) {
    String response
    System.out.print(s);  // print prompt
    System.out.flush();
    // read response
    return response;
}
public String promptAndRead (String s) {
    String response;
    this.write(s); //print prompt
    // read response
    return response;
}

 Says “send message to ‘this’ object”
private void writeAndFlush(String s) {
    System.out.print(s);
    System.out.flush();
}

public void write (String s) {
    this.writeAndFlush(s);
}

public String promptAndRead(String s) {
    String response;
    this.writeAndFlush(s);
    //other code
}

## Choosing: Printing Prompt

<table>
<thead>
<tr>
<th>Approach</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use two lines</td>
<td>Can see what is done</td>
<td>More lines</td>
</tr>
<tr>
<td>Use write method</td>
<td>Abstract description of what’s done</td>
<td>Can’t see precisely what is done</td>
</tr>
<tr>
<td>Use helper method</td>
<td>Abstract description of what’s done</td>
<td>More methods</td>
</tr>
</tbody>
</table>
Next Choice: Input

- Choice is how to deal with BufferedReader
- Local variable
  - Object created and destroyed each time read
  - Better if few prompts – saves space
- Instance variable
  - Object created once for each InteractiveIO object
  - Better if as many prompts as writes – time

Multiple method implementations require careful considerations of the effects upon the other class code and class execution.
public String promptAndRead(String s) throws Exception {
    String response;
    this.write(s);
    BufferedReader kb;
    kb = new BufferedReader(new InputStreamReader(System.In));
    response = kb.readLine();
    return response;
}
First Alternative Continued

- BufferedReader only exists in promptAndRead
- Constructor has nothing to do
- Looks like
  ```java
  public InteractiveIO () {
  }
  ```
class InteractiveIO {

public InteractiveIO() throws Exception {
    kb = new BufferedReader(
        new InputStreamReader(System.In));
}

// definition of write
// definition of promptAndRead
private BufferedReader kb;
}
public String promptAndRead(String s) throws Exception {
    String response;
    this.write(s);
    response = kb.readLine();
    return response;
}
Role of Constructor

- Constructor important when using instance variables
- Ensures that variables refer to an object
- Ensures that object referred to is correct one
- Methods can assume these things
- Methods should also ensure these assumptions will be true when done
class InteractiveIO {
    // creates object ready to send output to console
    public InteractiveIO() throws Exception {
        kb = new BufferedReader(
                new InputStreamReader(System.In));
    }
    // Writes string s to console
    public void write(String s) {
        System.out.print(s);
        System.out.flush();
    }
    // continued next slide
// Prompts using s, reads and returns response
public String promptAndRead(String s) throws Exception {
    String response;
    this.write(s);
    response = kb.readLine();
    return response;
}

private BufferedReader kb; // for keyboard input
Key Points

- Design by specifying interface
- Implement class by choosing how to implement methods and whether to use instance variables
- Use instance variables to remember objects/values that are needed in more than one method or method invocation
- Can change implementation without affecting behavior – behavior of object is what’s important
Implementation Strategy

- Even for programming languages like C best to think of what data is needed to implement program first
- For us this means deciding on what attributes an object has and using these as instance variables

Object-oriented programming is data centric.
Another Example

- Most classes motivated by need to model some real world entity or concept
- Instance variables can be attributes of object
- Example: a person’s name
- Attributes: first name, last name and title (optional)
- Note: if consider ethnicity, names may have other forms (e.g., hispanic names)
Name Class Behaviors

- Create name object
- Get initials as a String object
- Get name as String, first-last order
- Get name as String, last name, first order
- Add or update title
class Name {
    public Name(String first, String last) { … }
    public String getInitials () {  … }
    public String getLastFirst () { … }
    public String getFirstLast () { … }
    public void setTitle (String newTitle) { … }
    //instance variables here
}

Attributes of object are known:
- first name
- last name
- title

All are strings

```java
private String firstName;
private String lastName;
private String title;
```
Constructor

Builds a standard name – without title

```java
public Name (String first, String last) {
    firstName = first;
    lastName = last;
    title = ""; // not everyone has a title
}
```

Ensures every instance variable has a value
public String getInitials () {
    String fstInit = firstName.substring(0,1);
    String secInit = lastName.substring(0,1);
    return (fstInit.concat(secInit));
}

**Accessor** method – returns information
About Methods

- Notice that there are no methods to change the first or last name
- Only a method to change the title – *mutator* method
- User of class would handle changing name by creating a new Name object
- This is another design choice
State and Behavior

- State of an object – values of instance variables
  
  Name n1 = new Name("Ben", "Keller");
  Name n2 = new Name("Neb", "Rellek");

- State determines behavior
  
  n1.getInitials() // “BK”
  n2.getInitials() // “NR”
Responsibility

- Most choices have to do with which part of program is responsible for what behavior
- Examples:
  - Printing issues in promptAndRead
  - Ownership of BufferedReader in InteractiveIO
  - Reading in Name object from input
  - Writing out Name object to output
Outputting Objects

- Idea: provide output method for objects
- User does not have to access object data to output object
- Issues:
  - Format for object – many choices
  - Where the data is sent – PrintStream is good choice
Example

Name badboy = \texttt{new} Name(\texttt{"Gary"}, \texttt{"Bundy"});

badboy.print(System.out);
badboy.print(backupFile); //assuming PrintStream
public void print (PrintStream target) {
    target.print(title):
    target.print("");
    target.print(firstName);
    target.print("");
    target.print(lastName);
}

Inputting Objects

- For input it doesn’t make sense to ask a Name object to read another object.
- Instead make a request to Name class through a class method.
- Class method is declared in Java by including the word static before the return type in the declaration.

Objects will usually not exist before data to be stored in them is input.
Class Methods

- Class methods are not associated with objects
  - So
    - Must be invoked using class name instead of object reference
    - May not access any instance variables.
public static Name read(BufferedReader br) throws Exception {
    String first, last;
    first = br.readLine();
    last = br.readLine();
    return new Name(first, last);
}
Name nme;
// read from keyboard
BufferedReader kb;
kbl= new BufferedReader(
   new InputStreamReader(System.in));

nme = Name.read(kb); // no instance