Class Structure
Classes

• 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**
  – Similar to Python class methods
  – Will come back to these later
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`, `private` or no modifier before the word `class`
  – Default is `protected`
  – This basically defines the visibility of the class
  – `public` means any program in the project can use (i.e., create objects/call methods) that class
  – `protected` means only programs in the same package and subclasses in other packages can use the class
  – `no modifier` means only programs in the same package can use the class
  – `private` means only methods within that class can use the class

• Things in a class can be in any order

• Eclipse example: PezDispenser.java
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

• Eclipse example: PezDispenser.java
Defining constructors

- A constructor is code to create an object
  - Similar to Python’s `__init__` method
  - 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
- Eclipse example: PezDispenser.java
Defining a method

• A method has the syntax:

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

• Eclipse example: PezDispenser.java
Returning a result from a method

• If a method is to return a result, it must specify the type of the result
• You must use a return statement to exit the method with a result of the correct type
• Eclipse example: PezDispenser.java
Returning no result from a method

• The keyword **void** is used to indicate that a method doesn’t return a value
• The **return** statement must not specify a value
• There are two ways to return from a void method:
  – Execute a **return** statement
  – Reach the closing brace of the method
• Eclipse example: PezDispenser.java
Creating an object

• An object is also called an **instance of a class**
• You create an object of by calling its constructor and using the keyword **new**
• Syntax
  ```java
class ClassName{
  public ClassName(par1, ... parN){
    //some statements
  }
  public static void main(String[] args){
    ClassName c = new ClassName(arg1, ..., argN);
  }
}
```
• **Note:** **new** creates a new object on the heap
  – The object name is a pointer to that object (surprise, surprise)
• Eclipse example: PezDispenser.java
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:
  – object.method(extra information)
  – The object is the thing we are talking to
  – The method is a name of the action we want the object to take
  – The extra information is anything required by the method in order to do its job

• Eclipse example: PezDispenser.java
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;
}
```

• As with if-cases, you don’t need braces if your method is just one statement
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
```java
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();
    }
}
```
Methods and static methods

- Java has two kinds of methods: **static** methods and non-static methods (called **instance methods**)
  - Static methods are Java’s equivalent to class methods in Python
  - You call them using the class, not the object
  - They do not have access to instance variables
  - Most methods you write *should not* be static

- Every Java program has a **main** method, which is static

- Static methods should be used for operations that do not depend on instance variables or objects

- Eclipse example: Arithmetic.java
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
  - **null** is basically a pointer that doesn’t point to anything (with value 0)
  - Very similar to Python’s **None**
  - The error you will see is **NullPointerException**
Visibility

• Older languages greatly emphasized the notion of visibility
  – Especially Java and C/C++
• Visibility determines how accessible is your class/method/variable
  – Discussed class visibility earlier
  – Same idea for methods and variables
• Usually it is a good idea to make you instance variables private
  – Don’t want other people to arbitrarily change them
  – Instead, expose methods for modifying instance variables
• Can also have private methods
  – You don’t want anyone but you to use these – they are for some internal computations in your program
  – Will see examples of these later in the class
• Eclipse example: PezDispenserPrivate.java
Java style

• As with Python, the more important thing is to be consistent
  – Anyway, here are some “Java conventions”

• Naming is camelCase
  – Variable names start with a lowercase letter
  – Class names start with a capital letter

• Indentation is 2 or 4 spaces (no consensus here)
  – I think 4 looks better (but won’t take off points for 2 as long as you are consistent)
  – In Java indentation is style, so make sure you indent!

• Braces go on the same line as the method/loop definition
  – In C, they go on the next line on their own

• As always, you should have many comments