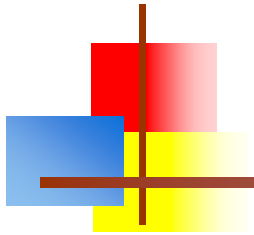
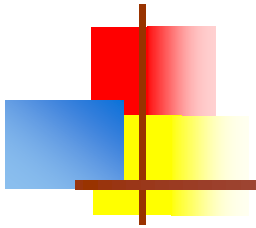


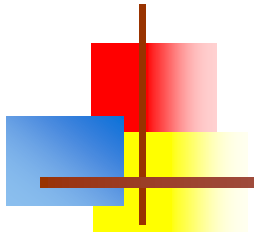
Access to Names

Namespaces,
Scopes,
Access privileges



Overview

- In Java you name various things: classes, methods, variables, etc.
- Sometimes you can refer to these things by name, but other times Java gives you an error
- You need to know when you *can* refer to something by name, and when you *can't*
- You also need to know *how* to refer to things
- Java's rules are complex, but they are not arbitrary--once you understand them, they *do* make sense!



Part I: Namespaces

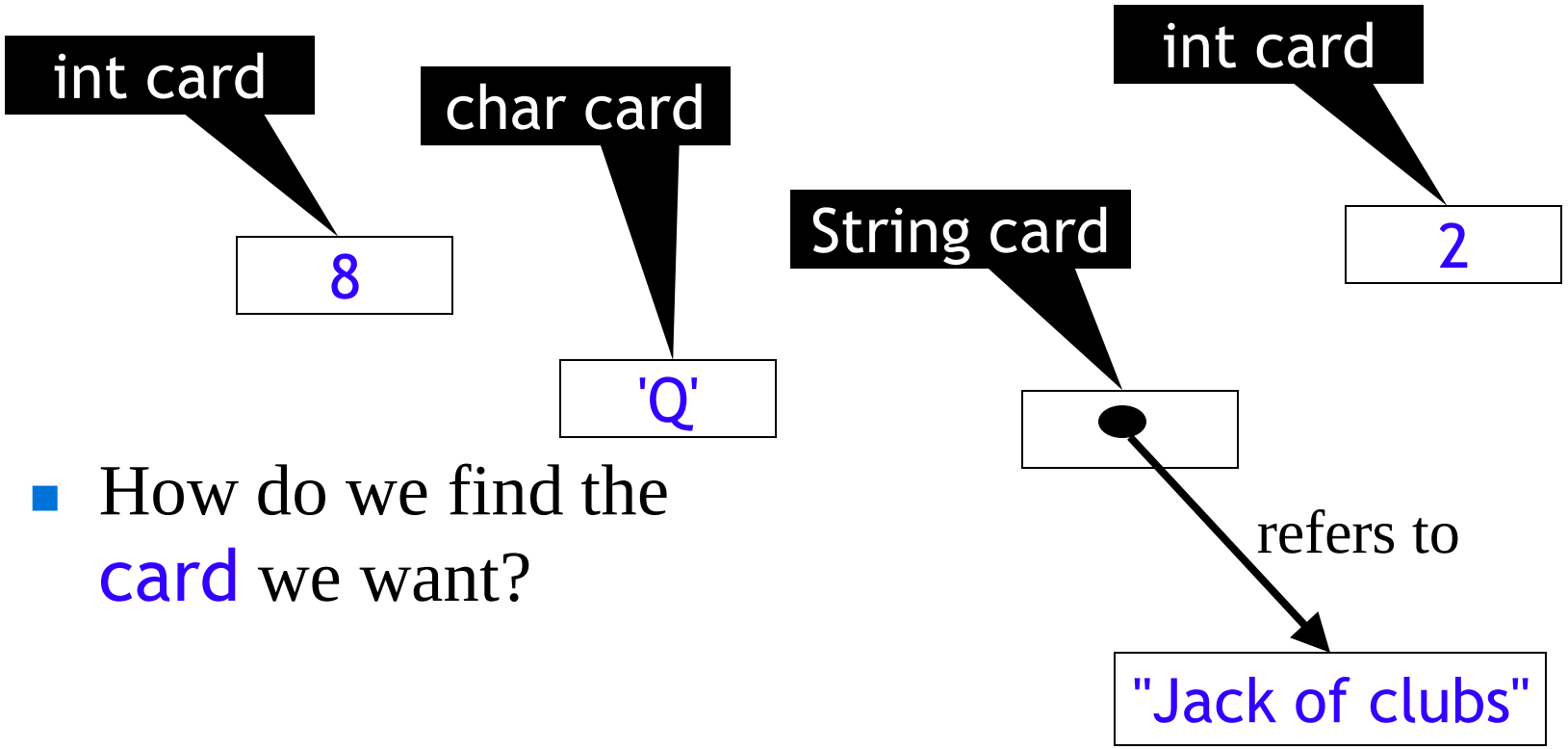
Names are not unique

Hi. My name is Mike Smith

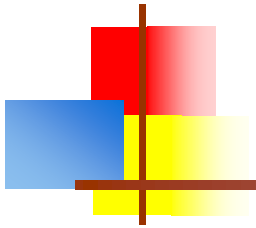
My name is Mike Smith

Hello. I'm Mike Smith

Variable names



- How do we find the card we want?



Declarations

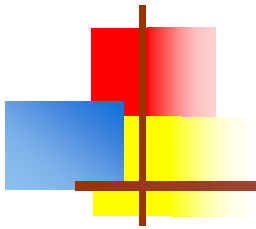
- Variables are declared like this:

  [*access*] [static] *type name* [= *value*] , ... ;

- Examples:

```
int m;
public double e = 2.718281828459045;
static final int ONE = 1, TWO = 2, THREE = 3;
public static boolean pluggedIn;
```

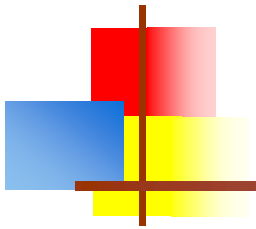
- Once we declare some variables, where can we use them?
- Java's rules are quite complex, but it's very important to understand them



Declare a variable *only once*

```
public class Test {
    public static void main(String[] args) {
        int var = 5;
        double var = 8.33;
        System.out.println(var );
    }
}
```

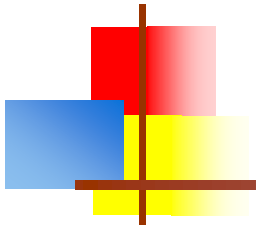
- var is already defined in main(java.lang.String[])



A little puzzle

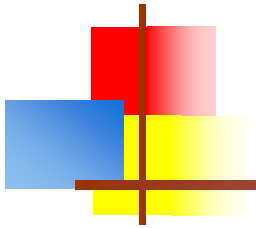
```
public class main {
    int main = 5;
    public static void main(String[] args) {
        main main = new main();
        System.out.print(main);
    }
}
```

- This is a legal program (!); what does it print?
- Answer: main@ecf76e
- Next question: why?



Namespaces

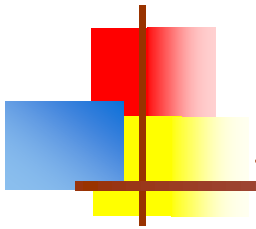
- Java figures out what kind of thing a name refers to, and puts it in one of six different namespaces:
  - package names
  - type names
  - field names
  - method names
  - local variable names (including parameters)
  - labels



The puzzle solved

```
public class main {    // type name
    int main = 5;  // field name
    public static void main(String[] args) { // method name
        main main = new main(); // local names (incl. args)
        System.out.print(main);
    }
}
```

- Java prints out object main@ecf76e in local variable main
- We haven't talked about package names or labels
- Note that this is ***terrible*** style!

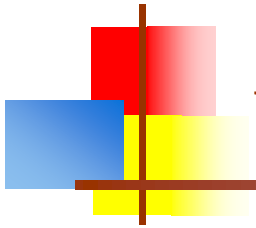


Another little puzzle

```
public class Test {
    static int five() { return 5; }
    public static void main(String[] args) {
        System.out.print(five);
    }
}
```

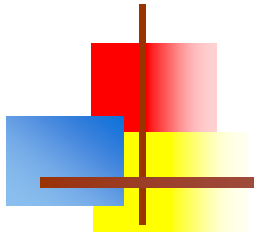
```
cannot resolve symbol
symbol :variable five  location: class Test
```

- Answer: five() is a method, but five looks like a local variable

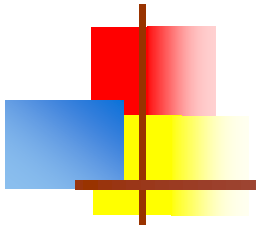


What you should remember

- A namespace is a place that Java keeps track of names
- Java uses six different namespaces
- If you name things intelligently, and don’t use the same name for different things, you don’t have to worry much about namespaces

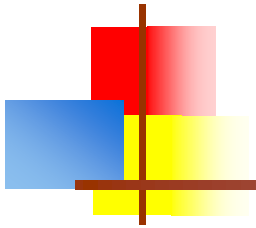


Part II: Scope

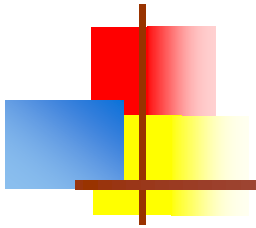
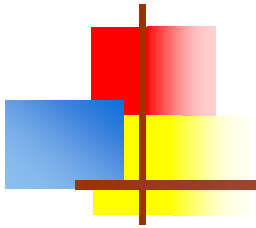
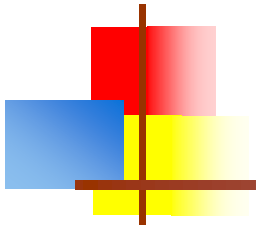
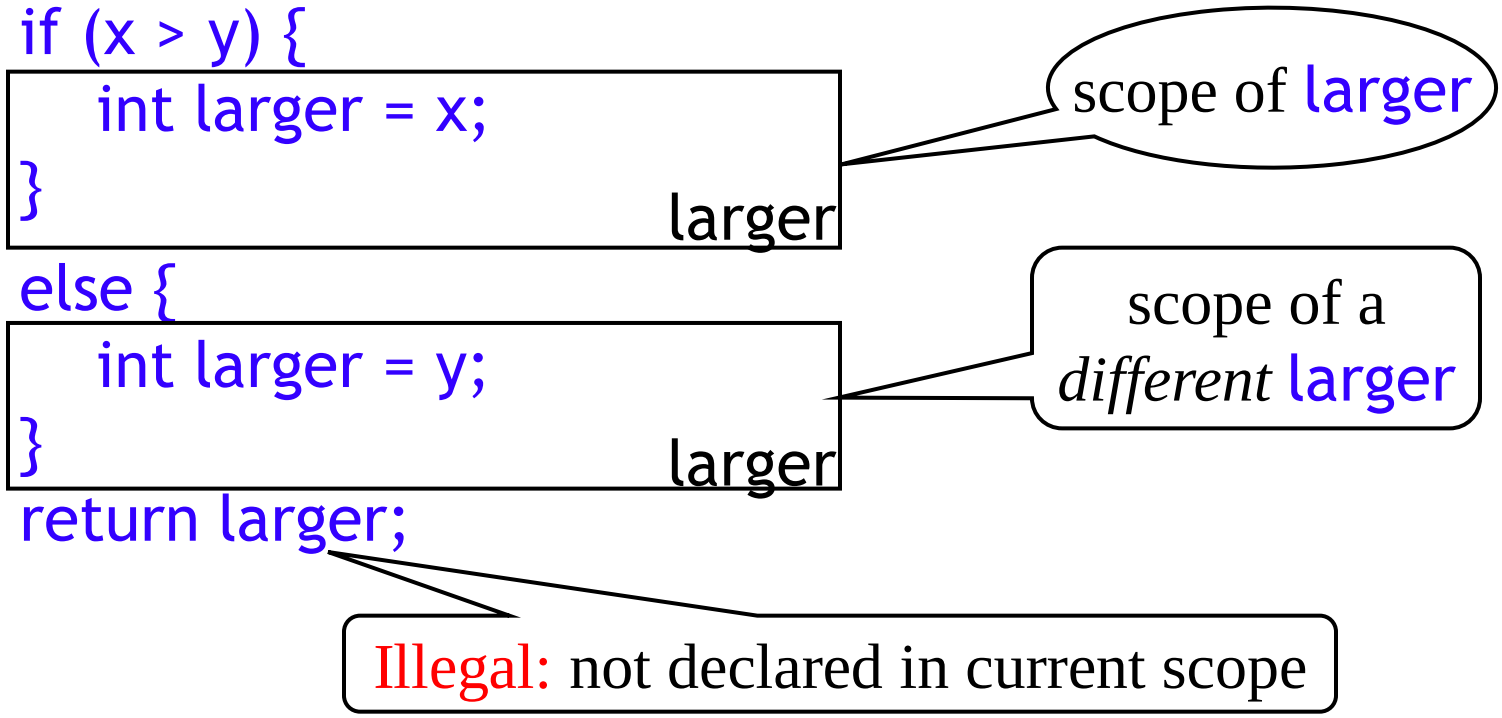
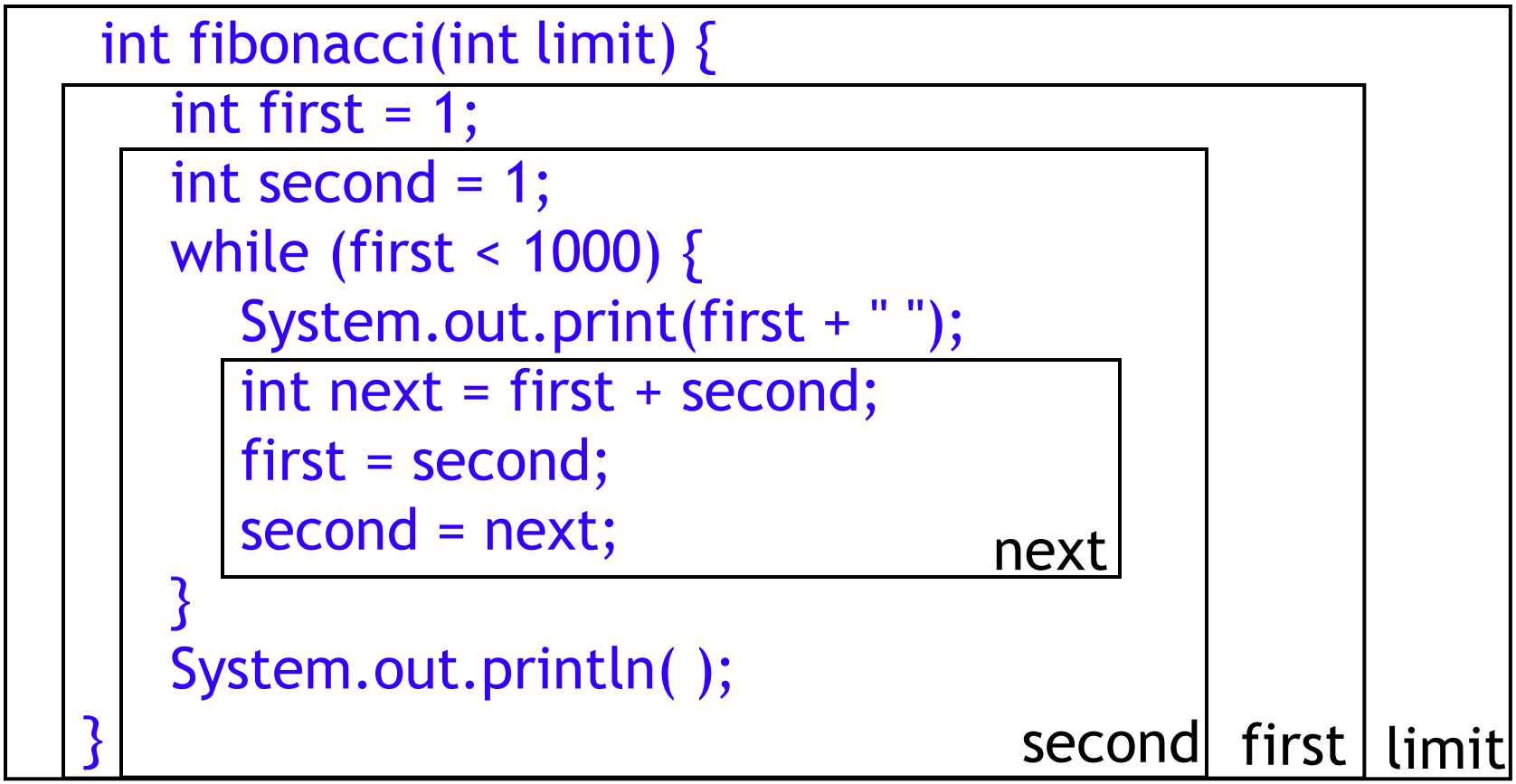
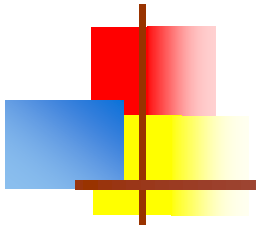


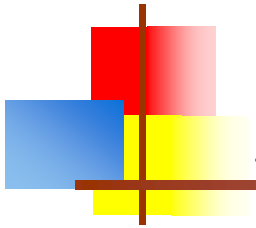
Scope

- The scope of a name is the part of the program in which the name is visible
- In Java, scope rules apply to single methods
- Variables declared in a method can *only* be used within that method; you cannot *ever* use them anywhere outside the method
- Between classes, we use *access rules* rather than scope rules

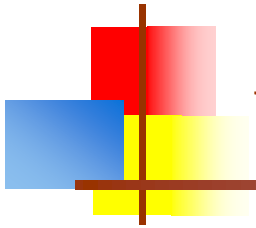
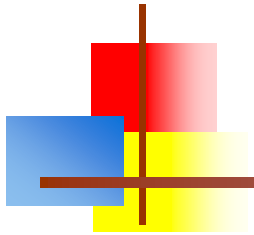
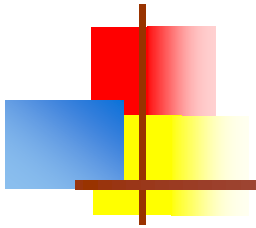
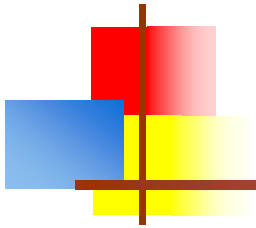
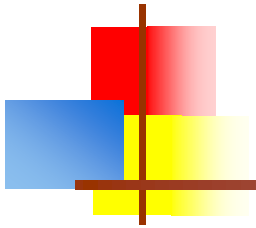
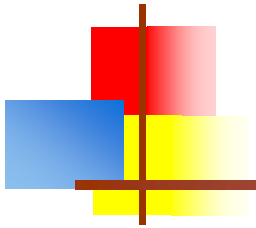
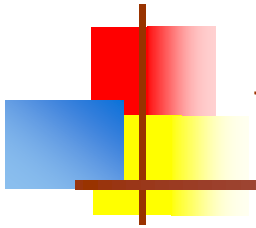
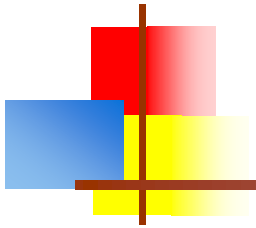
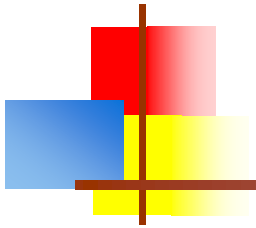


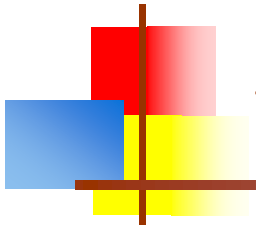
Methods may have local variables

- 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
    ```



Compound statements and blocks

- A compound statement consists of zero or more statements inside braces
  - Examples: `{ }` , `{ temp = x; x = y; y = temp; }`
- A block consists of zero or more statements *or declarations* inside braces
  - Example: `{ int temp = x; x = y; y = temp; }`
- This distinction is **not** important in Java
- I'll just use the terms interchangeably



Blocks occur in methods

- The braces in a class declaration do *not* indicate a compound statement:

```
public class MyClass { /* not a block */ }
```

- Elsewhere, braces *do* indicate a compound statement:

```
int absoluteValue(int n) {
    if (n < 0) {
        return -n;
    }
    else return n;
}
```



Declarations in a class

- The braces in a class declaration do *not* indicate a block or compound statement:

```
public class MyClass { // not a block
    int foo;               // instance variable
    static int bar;        // class variable
```

- Instance variables and class variables are available *throughout the entire class* that declares them
  - Java doesn't care in what order you declare things
    - However, declarations *with initializations* must precede use of their value
      - Example:

```
int half = whole / 2;
int whole = 100;
```

        is ***not*** legal
  - It's usually *good style* to put variable declarations first, then constructors, then methods

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*



Nested scopes





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

```
void multiplicationTable() {
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++)
            System.out.print("  " + i * j);
        System.out.println();
    }
}
```

j

i

Duplicate definitions

```
void duplicate1( ) {
    int i = 0;
    for (int i = 0; i < 10; i++) { //illegal
        System.out.println(i);
    }
}

void duplicate2( ) {
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    for (int i = 0; i < 10; i++) { // legal
        System.out.println(i);
    }
}
```

```
public class Scope {
    int i;
    void duplicate3( ) {
        for (int i = 0; i < 10; i++) { // legal
            System.out.println(i);
        }
    }
}
```



A common error

```
class Something {
    String example;

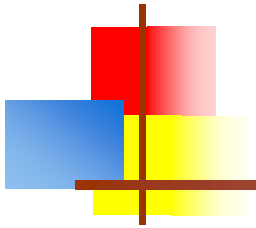
    public static void main(String[ ] args) {
        String example = "xyz";
        testPrint();
    }

    static void testPrint() {
        System.out.println(example);
    }
}
```

- Output: null
- Why?
  - Local variables shadow class variables with the same name
  - The problem is harder to notice in a longer program



What you should remember

- Names (of variables, constructors, or methods) declared *anywhere* in a class are available *everywhere* within the class (order doesn't matter)
- Formal parameters of a method are available *everywhere* within the method
- Variables declared in a block are available *from where they are declared* to *the end of that block*
- Variables declared in a `for` loop are available *within the `for` loop*



Part III: Access privileges



Packages = directories = folders

- A public class must be put in a file of the same name
  - Example: `public class Test { ... }` must be saved in a file named `Test.java`
- Similarly, if you use a `package` statement, the file must be in a *directory* (folder) of the same name
  - Example: If you specify `package assignment_2;` then it must be in a directory named `assignment_2`
- Why use more than one package in a program?
  - Sometimes you want to write classes that are useful in many different programs
  - Sometimes you may be working on a large program that needs the extra level of organization
    - We aren't writing large programs in this course



Scope and access

- Local variables (formal parameters and method variables) are available only within the method that declares them, *never anywhere else*
- Names (of variables, constructors, and methods) declared in a class are available *everywhere within that class*, and *may* be available inside *other* classes
  - Access to these names is controlled by the access modifiers public, package (default), protected, and private



How to access names

- From outside class Person:
  - you can access an *instance variable* (say, of jack) by:

    ```
    jack.age
    ```
  - you can access a *class variable* by:

    ```
    Person.population
    ```
- As a (confusing) convenience, you can also access a *class variable* by way of any *instance* of that class:

  ```
  jack.population  // works, but is confusing--avoid
  ```
- These techniques also work for methods and constructors



public and private access

- If you declare a name to be public, you are allowing every other class in the world to see it and to change it (called read-write access)
  - If random changes to this name can invalidate the object, *it should not be public*
- If you declare a name to be private, you are saying that only the class in which it is declared can see it and change it
- If all your `.java` files are in the same directory (recommended for this course), there is no difference between public, protected, and package



Why private is important

- The fields (instance variables) of an object describe its state
  - This is just about the *only* legitimate use of instance variables
  - Other communication between methods should be done with parameters
- The state of an object must be kept ***valid***
  - Examples: Employee IDs must be unique; a person's age may not be negative; a tic-tac-toe game may contain Xs and Os, but not Ms
  - From outside the class, objects must *always* be valid
  - Inside the class, objects may be *temporarily* in an invalid state, as they are being manipulated
- It is the responsibility of a class to ensure that objects of that class are, and remain, valid
  - If a field is not private, the object can be manipulated from outside the class, and the class loses control
- Moral: ***Instances variables should almost always be private***



Package and protected access

- Package access means that a name is available everywhere in the same package (the same directory)
- protected access means that a name is available everywhere in the same package (the same directory), but also to any subclasses, wherever they may be
- protected access is "more public" than package access

- Question: Why have protected access?
- Answer: Because, although you would usually prefer your instance variables to be private, sometimes you need to access them in subclasses
  - It would be nice if protected variables were available in subclasses but not to everything in the same directory
  - Access controls in Java are not very well designed ☹



Read-only access

- If you want a variable to be read-only:
  - Declare the variable to be private
  - Provide a "getter" method to return its value
  - *Do not* provide a "setter" method to set its value
- Example:
  - ```
    public class Person {
        private int population;
        int getPopulation( ) { return population; }
        …
    }
    ```
    ```



Vocabulary

- namespace -- a place that Java keeps track of names
- scope of a name -- the part of the program in which the name is visible
- compound statement -- zero or more statements inside braces
- block -- zero or more statements *or declarations* inside braces
- access modifier -- one of the keywords public, protected, and private



The End