Computer Science 22: Object Oriented Programming

Lecture #9: Encapsulation

In this Lecture

- Encapsulation
- Interface and Implementation
- Encapsulation: Methods and Attributes
- The Access Modifiers

Encapsulation

- "Information Hiding"
  - … but information hiding is a means of achieving *encapsulation*
- The process of keeping/hiding "secrets" of an object that do not contribute to its essential characteristics
  - "secrets" include both **structure** (data) and **implementation**
- Serves **to separate the interface of abstraction from its implementation**

Encapsulation Analogy

Encapsulation Analogy

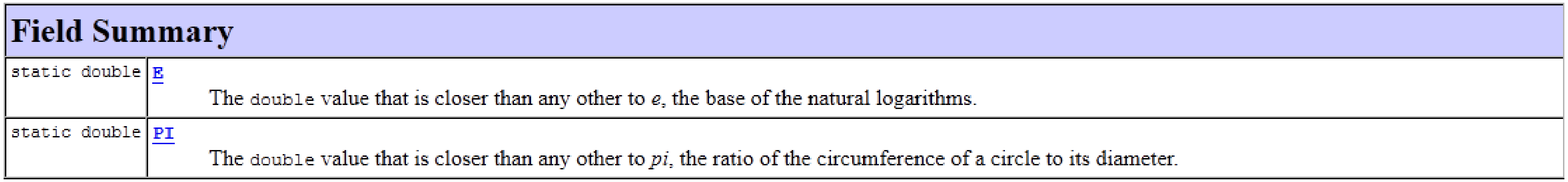
Classes and Objects

- Classes and objects are defined by their **interface** and by their **implementation**
  - **Interface** of a class captures the outside view, from which we can assert all assumptions
  - **Implementation** comprises the representation of the abstraction (i.e., data/data structures) and the mechanisms required to achieve the desired behavior

Interface: Java API Classes

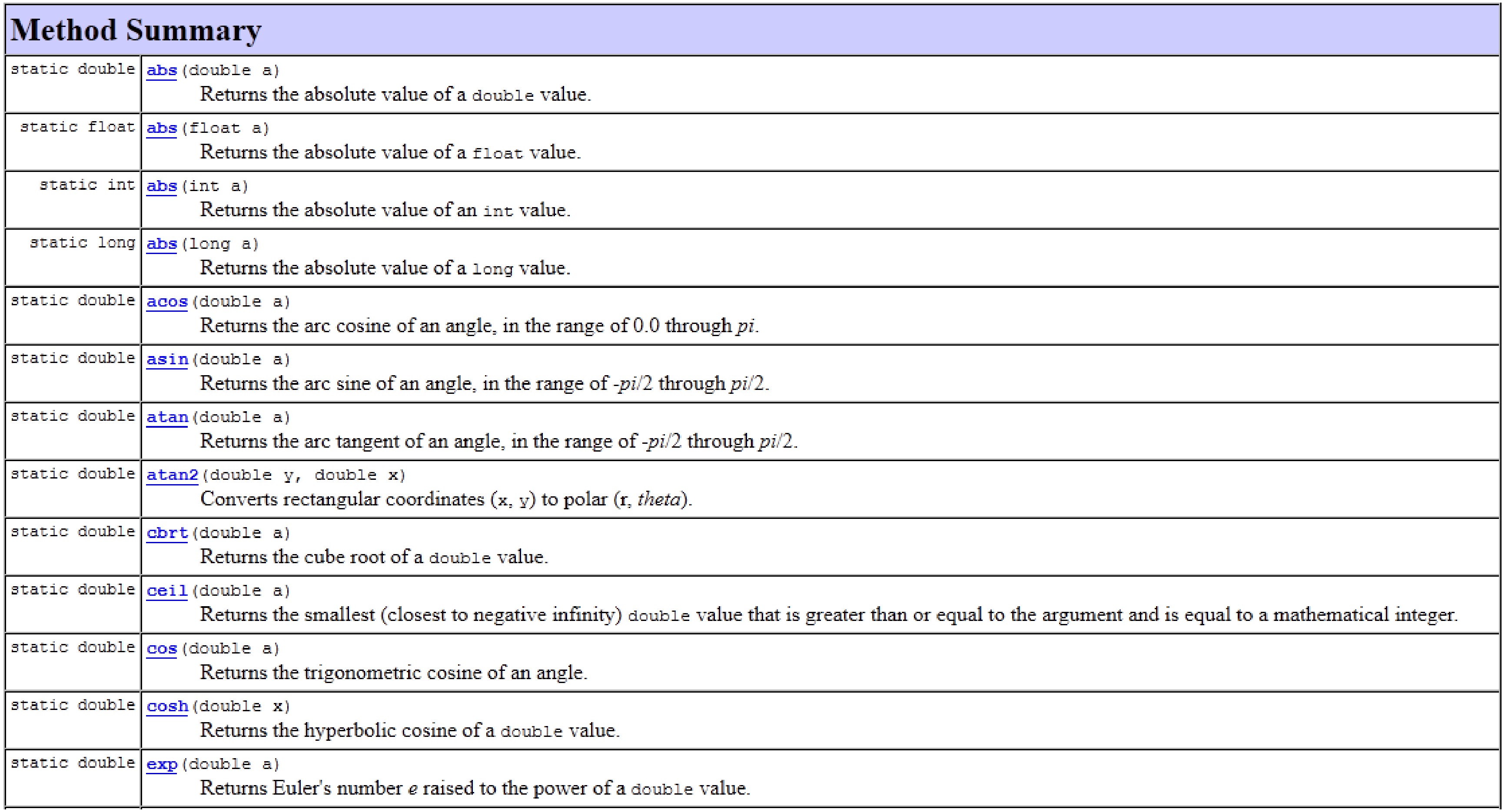
- Documented in the Java API Specification are the “available” **attributes** and **methods** that we can do with the classes

Interface: Math Class



Field Summary	
`static double`	**E** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**PI** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Interface: Math Class



Method Summary	
`static double`	`abs(double a)` Returns the absolute value of a `double` value.
`static float`	`abs(float a)` Returns the absolute value of a `float` value.
`static int`	`abs(int a)` Returns the absolute value of an `int` value.
`static long`	`abs(long a)` Returns the absolute value of a `long` value.
`static double`	`acos(double a)` Returns the arc cosine of an angle, in the range of 0.0 through *pi*.
`static double`	`asin(double a)` Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.
`static double`	`atan(double a)` Returns the arc tangent of an angle, in the range of -*pi*/2 through *pi*/2.
`static double`	`atan2(double y, double x)` Converts rectangular coordinates (`x`, `y`) to polar (r, *theta*).
`static double`	`cbrt(double a)` Returns the cube root of a `double` value.
`static double`	`ceil(double a)` Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer.
`static double`	`cos(double a)` Returns the trigonometric cosine of an angle.
`static double`	`cosh(double x)` Returns the hyperbolic cosine of a `double` value.
`static double`	`exp(double a)` Returns Euler's number *e* raised to the power of a `double` value.

Implementation: Math Class

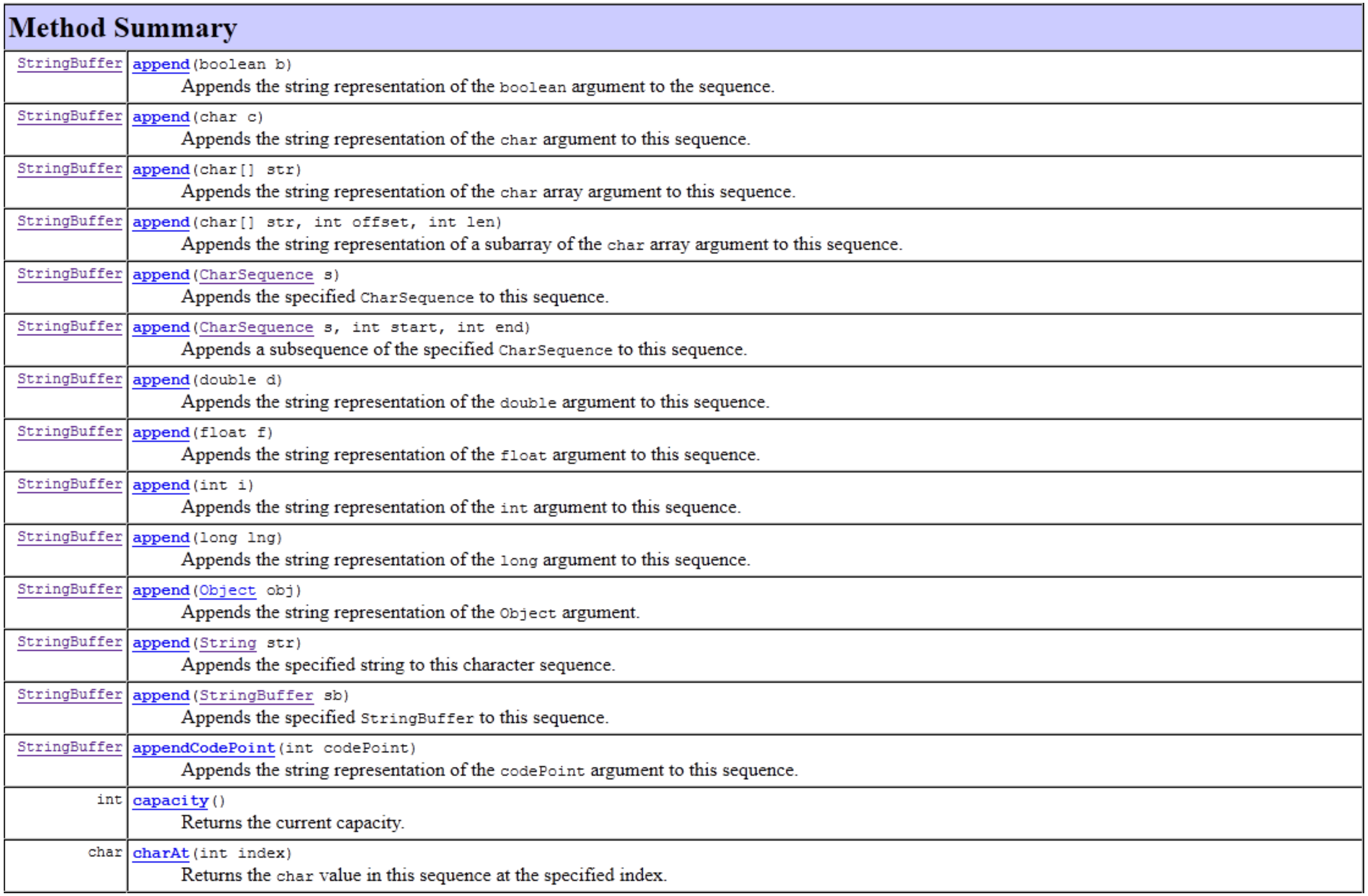
???

Interface: StringBuffer Class

No visible/accessible attributes.

No interface to StringBuffer attributes

Interface: StringBuffer Class



Method Summary	
StringBuffer	append(boolean b) Appends the string representation of the boolean argument to the sequence.
StringBuffer	append(char c) Appends the string representation of the char argument to this sequence.
StringBuffer	append(char[] str) Appends the string representation of the char array argument to this sequence.
StringBuffer	append(char[] str, int offset, int len) Appends the string representation of a subarray of the char array argument to this sequence.
StringBuffer	append(CharSequence s) Appends the specified CharSequence to this sequence.
StringBuffer	append(CharSequence s, int start, int end) Appends a subsequence of the specified CharSequence to this sequence.
StringBuffer	append(double d) Appends the string representation of the double argument to this sequence.
StringBuffer	append(float f) Appends the string representation of the float argument to this sequence.
StringBuffer	append(int i) Appends the string representation of the int argument to this sequence.
StringBuffer	append(long lng) Appends the string representation of the long argument to this sequence.
StringBuffer	append(Object obj) Appends the string representation of the Object argument.
StringBuffer	append(String str) Appends the specified string to this character sequence.
StringBuffer	append(StringBuffer sb) Appends the specified StringBuffer to this sequence.
StringBuffer	appendCodePoint(int codePoint) Appends the string representation of the codePoint argument to this sequence.
int	capacity() Returns the current capacity.
char	charAt(int index) Returns the char value in this sequence at the specified index.

Implementation: StringBuffer Class

???

Encapsulation

- The structure of an object as well as the **implementation** of behavior/methods are **hidden**
- Only the **interface** of an object is made **accessible**

Encapsulation

- The process of compartmentalizing the elements of an abstraction that constitutes its structure and behavior; it serves to separate the contractual interface of an abstraction and its implementation

Questions on Encapsulation

- What is important to the “user” of the instance?
  - User = may be another object/class
  - What operations/methods/data should you expose to the user of the instance?

Making Things Accessible and Not

- Consider the following structural/compositional hierarchy
  - Packages contain classes (also subpackages)
- Classes contain
  - Attributes/Variables (can have access modifiers)
  - Methods (can have access modifiers)

Making Things Accessible

- In Java Programming, we introduce certain keywords to the attributes/variables and methods of our class to define their level of accessibility:
  - public (**+**)
  - private (**-**)
  - protected (**#**)
  - *none* (package-default) (**~**)
- These keywords are called **access modifiers**

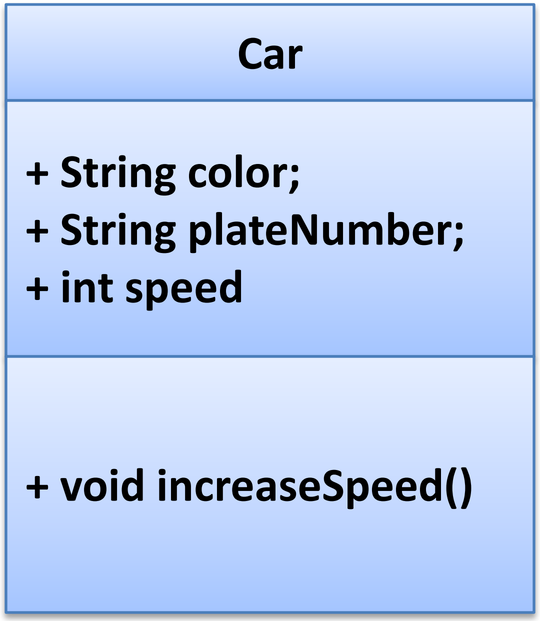
Public (+)

- Can be used in instance variables, methods, and class in package
- It means the attribute or method is accessible to ALL users
- The **public** attributes and methods of a class defines the **interface** of the class

Public (+)

```
public class Car {
   public String color;
   public String plateNumber;
   public int speed;

   public void increaseSpeed(){
        //codes
   }
}
```



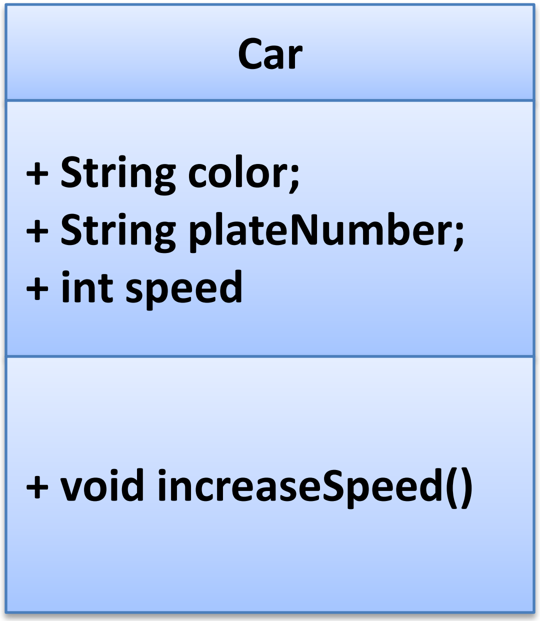
Car

+ String color;
+ String plateNumber;
+ int speed

+ void increaseSpeed()

Public (+)

```
public class CarExample {
   public static void main(String[] args){
        Car c = new Car();
        c.plateNumber = 2;    //valid
        c.color = “Red”;      //valid
        c.increaseSpeed();    //valid
   }
}
```



Car

+ String color;
+ String plateNumber;
+ int speed

+ void increaseSpeed()

Private (-)

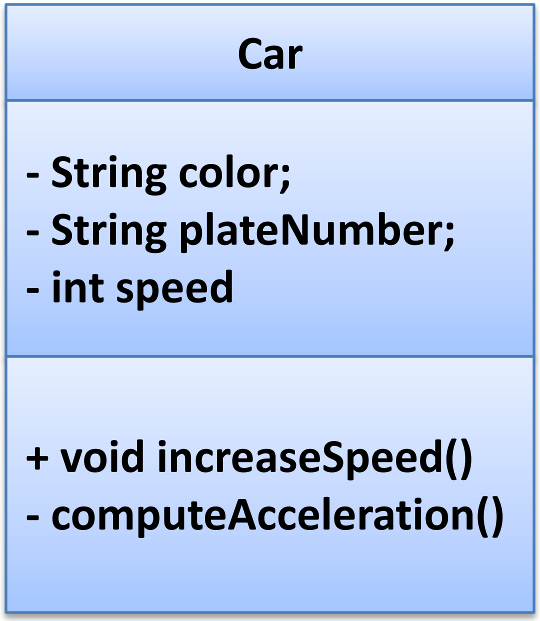
- Can be used in methods and instance variables only
- A private attribute/variable or method is accessible only by the class (i.e., private to the class)
- A private attribute/variable or method is NOT accessible by instances of other classes

Private (-)

```
public class Car {
   private String color;
   private String plateNumber;
   private int speed;

   public void increaseSpeed(){
        // call to method inside the same class
        int a = computeAcceleration();
        speed += a;
   }

   private int computeAcceleration(){
        // complex code, laws of physics, etc, etc
   }
}
```



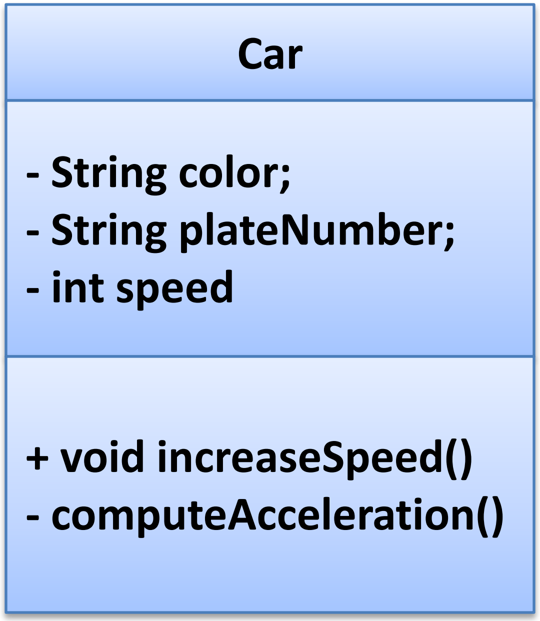
Car

- String color;
- String plateNumber;
- int speed

+ void increaseSpeed()
- computeAcceleration()

Public (-)

```
public class CarExample {
    public static void main(String[] args){
        Car c = new Car();
        c.plateNumber = 2;        //not valid
        c.color = “Red”;          //not valid
        c.increaseSpeed();        //valid
        c.computeAccelaration(); //not valid
    }
}
```



Car

- String color;
- String plateNumber;
- int speed

+ void increaseSpeed()
- computeAcceleration()

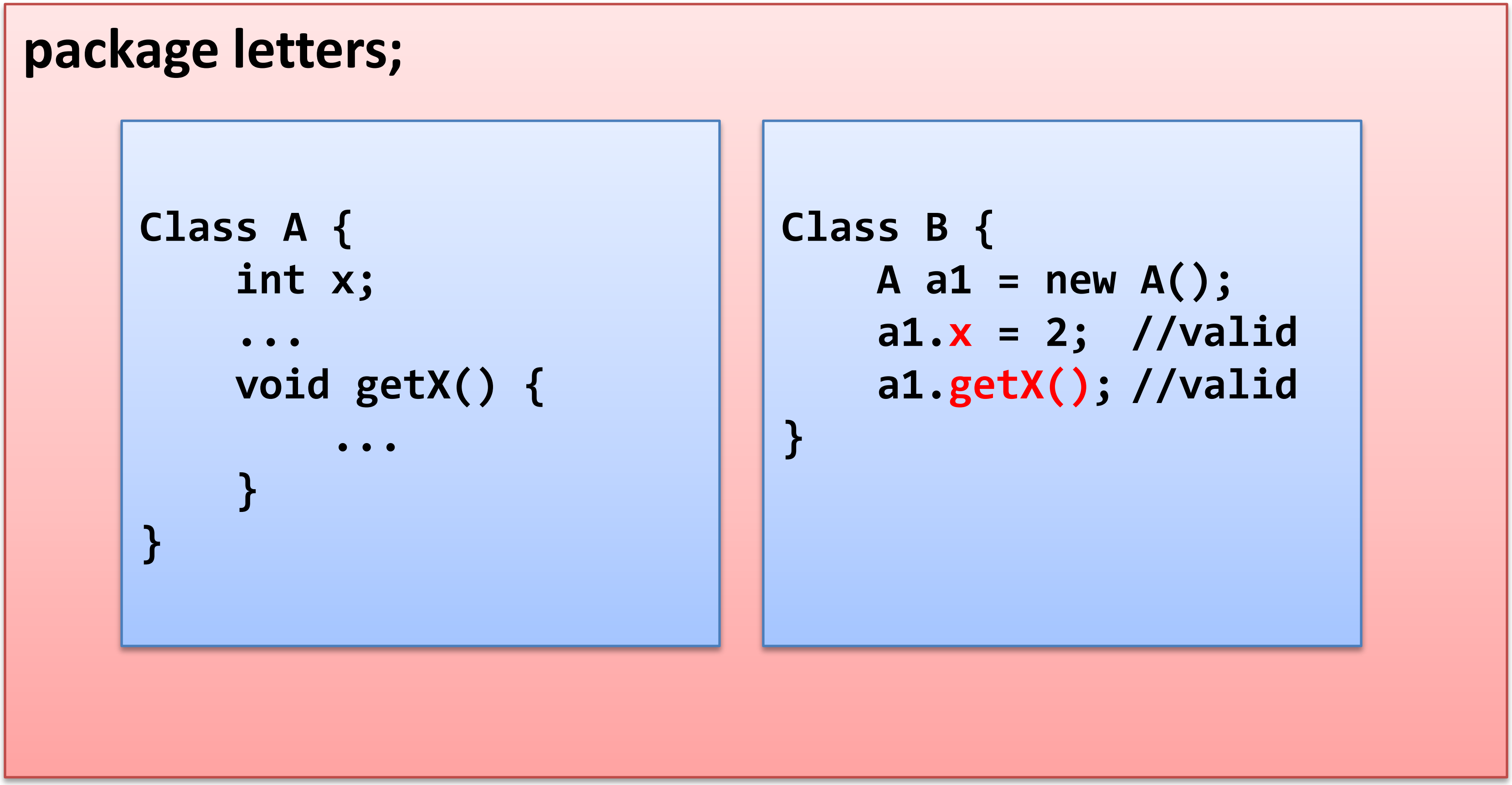
Protected (#)

- Attributes/variables and methods of a class that are protected are those accessible by subclasses of the that class
- More on this when we discuss inheritance

Package-Default (~)

- Package-default attributes/variables and methods are accessible to other classes so long as the class containing them and the classes that will use them belong to the same package.

Package-Default (~)



package letters;

```
Class A {
  int x;
  ...
  void getX() {
    ...
  }
}
```

```
Class B {
  A a1 = new A();
  a1.x = 2;  //valid
  a1.getX(); //valid
}
```

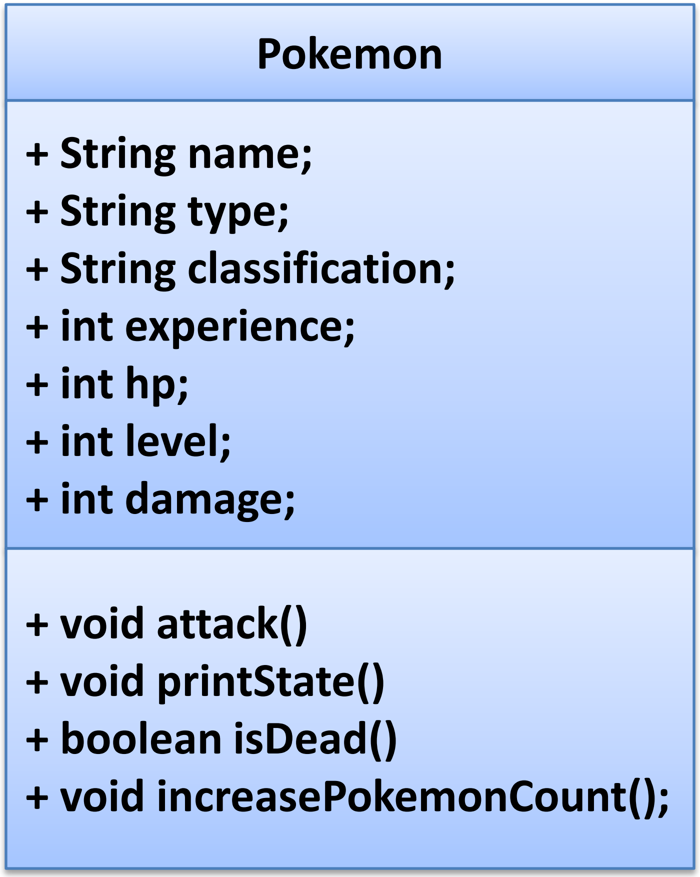
Encapsulation

Why bother with encapsulation?

The “damage” Attribute of a Pokemon

- The damage that a pokemon can deal only increase when the pokemon levels up

Notations, notations



Pokemon

+ String name;
+ String type;
+ String classification;
+ int experience;
+ int hp;
+ int level;
+ int damage;

+ void attack()
+ void printState()
+ boolean isDead()
+ void increasePokemonCount();

- Pokemon Class with all the attributes/variables and methods declared as **public**

The “damage” Attribute of a Pokemon

- Let’s assume we used an **int** for damage and for some reasons we declared this attribute public (or accessible to all)
- Then to access damage,
  - **bulba.damage**
  - What happends if at the middle of “execution” we do: **bulba.damage = 1000000**;
  - Don’t you think this is a BIT unfair for pika?

The Dangers of Exposing Things

- If we make “damage” public then we are allowing the user of bulba to modify the structure at will.
- This breaks the logic of the system (i.e., *The damage that a pokemon can deal only increase when the pokemon levels up*)!

How About These?

- pika.hp = 0; // A BIT unfair for pika?
- pika.level = -7; // A BIT unfair for pika?

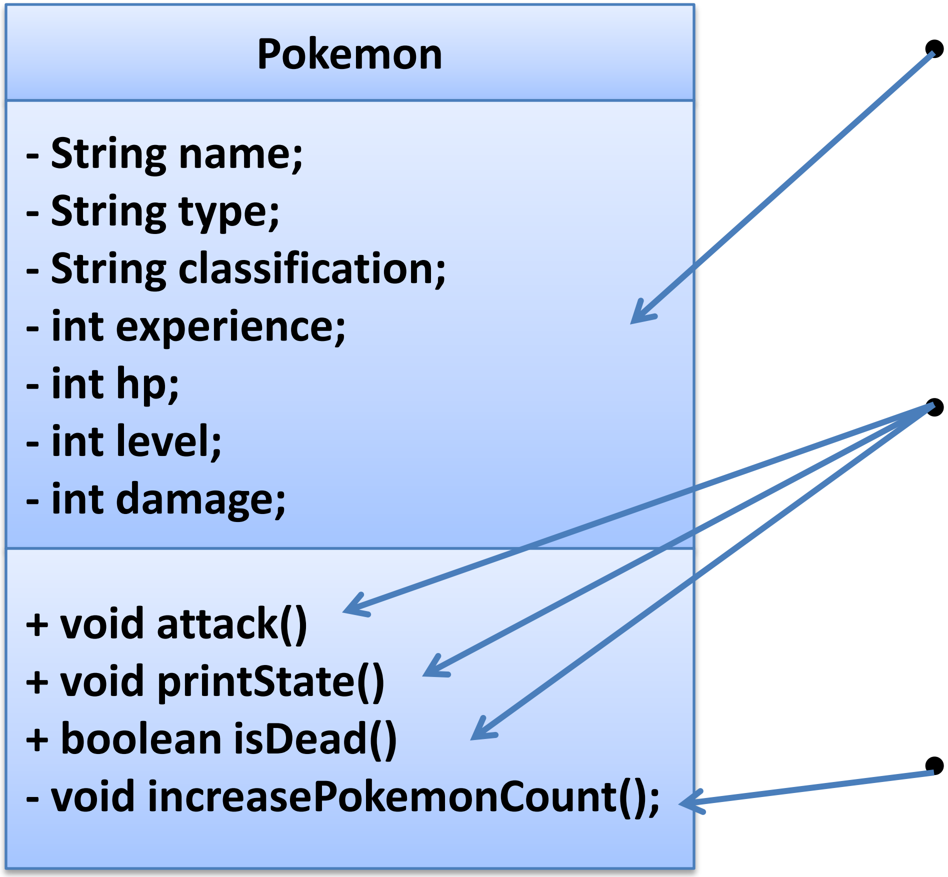
Hiding the Implementation

- By defining/describing the result of the behavior in the interface, we lessen the complexity of our class and lessens the concerns of the user of the class.
- How did we implement **attack()**?

Hiding the Implementation

```
public void attack(Pokemon enemyPokemon){
    /*
    You can write the most complex of codes here
    as long as the end result of this method does
    what it says it does. You can also change how
    this is implemented without much effect to
    the users,
    */
}
```

And So



```
Pokemon

- String name;
- String type;
- String classification;
- int experience;
- int hp;
- int level;
- int damage;

+ void attack()
+ void printState()
+ boolean isDead()
- void increasePokemonCount();
```

- In most classes, the attributes/variables are usually hidden/private
- Only operations that are allowed for users are exposed
- Private methods are used “internally”

Encapsulation in Other Languages

- The notion of public, private, and protected can be found in almost all OO programming languages

Summary

- Encapsulation means hiding details not relevant OR does not contribute to the essential characteristics of the object
- Encapsulation means protecting your class/object from misusage or “accidents”
- Encapsulation makes abstraction work