Chapter 15: Object Oriented Programming

Think Java:
How to Think Like a Computer Scientist

5.1.2

by Allen B. Downey

How do Software Developers use OOP?

- Defining classes – to create objects
- UML diagrams – to capture essence of class
- Javadoc – to document code
- Inheritance – to reuse code

Agenda

- Review of objects and classes 🡐
  - standard class pattern
  - UML diagrams
  - toString, equals
  - Javadoc documentation
- OOP's Big 3 Concepts:
  - Encapsulation
  - Inheritance
  - Polymorphism

Standard classes

- We have learned how to write classes that create objects
- Some of the principles:
  - instance variables represent state of object
    - make them private
  - constructors allow you to create objects
    - overloading for multiple options
  - methods represent what you can do with object
    - make them public (in general…sometimes private)
    - Accessor methods – get information out of object
    - Modifier methods – change information inside object

Example: `Tile` class

- A `Tile` object has the following fields:
  - `letter`. The letter on the tile
  - `value`. The value of the tile

```
public class Tile
{
    private char letter;
    private int value;
}
```

Access Specifiers

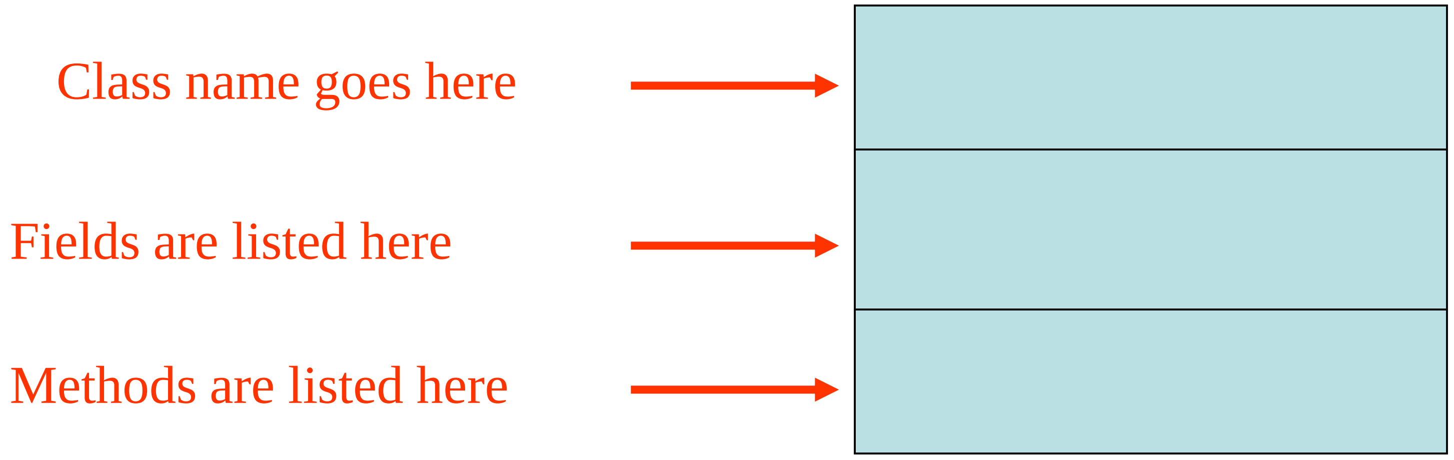
- An access specifier is a Java keyword that indicates how a field or method can be accessed.
- `public`
  - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
  - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

The `Tile` class methods

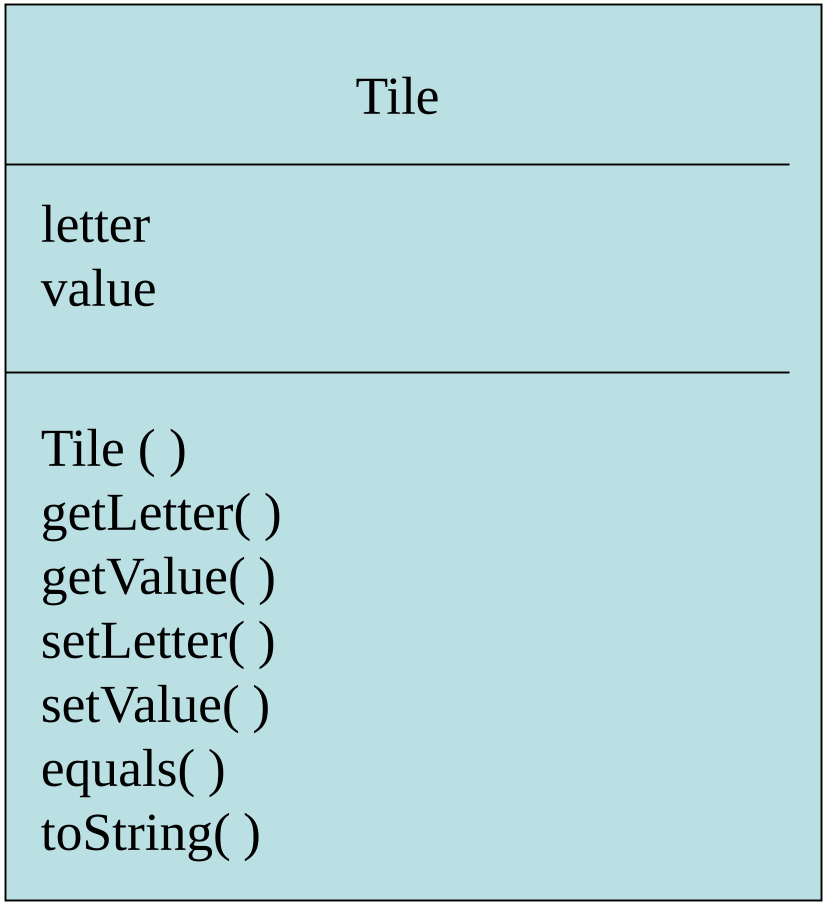
- The `Tile` class also has the following methods:
  - **`Tile  (constructor – 2 of them)`**
  - **`getLetter`**
  - **`getValue`**
  - **`setLetter`**
  - **`setValue`**
  - **`equals`**
  - **`toString`**

UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.



UML Diagram for `Tile` class



Header for the `setLetter` Method

Return Type

Access specifier

Method Name

Notice the word **`static`** does not appear in the method header designed to work on an instance of a class (instance method).

```
public void setLetter (char letter)
```

Parameter variable declaration

Writing the `setLength` Method

```
/**
   The setLetter method stores a value in the
   letter field.
   @param letter The value to store in letter field.
*/
public void setLetter(char letter)
{
   this.letter = letter;
}
```

Accessor and Modifier Methods

- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called *accessors*.
- The methods that modify the data of fields are called *modifiers*.
- Each field that the programmer wishes to be viewed by other classes needs an accessor.
- Each field that the programmer wishes to be modified by other classes needs a modifier.

Accessors and Modifiers

- For the `Tile` example, the accessors and modifiers are:
  - **`getLetter`** : Returns the value of the `letter` field.

    `public char getLetter()` …
  - **`getValue`** : Returns the value of the `value` field.

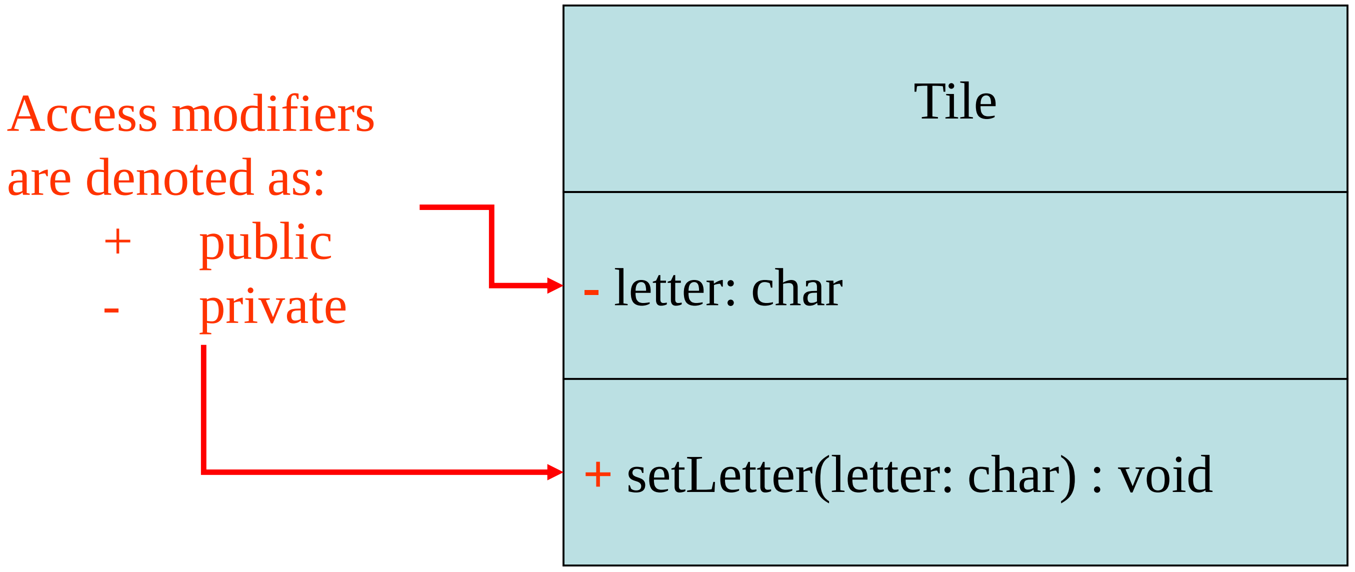
    `public int getValue()` …
  - **`setLetter`** : Sets the value of the `letter` field.

    `public void setLetter(char letter)` …
  - **`setValue`** : Sets the value of the value field.

    `public void setValue(int value)` …
- Other names for these methods are *getters* and *setters*.

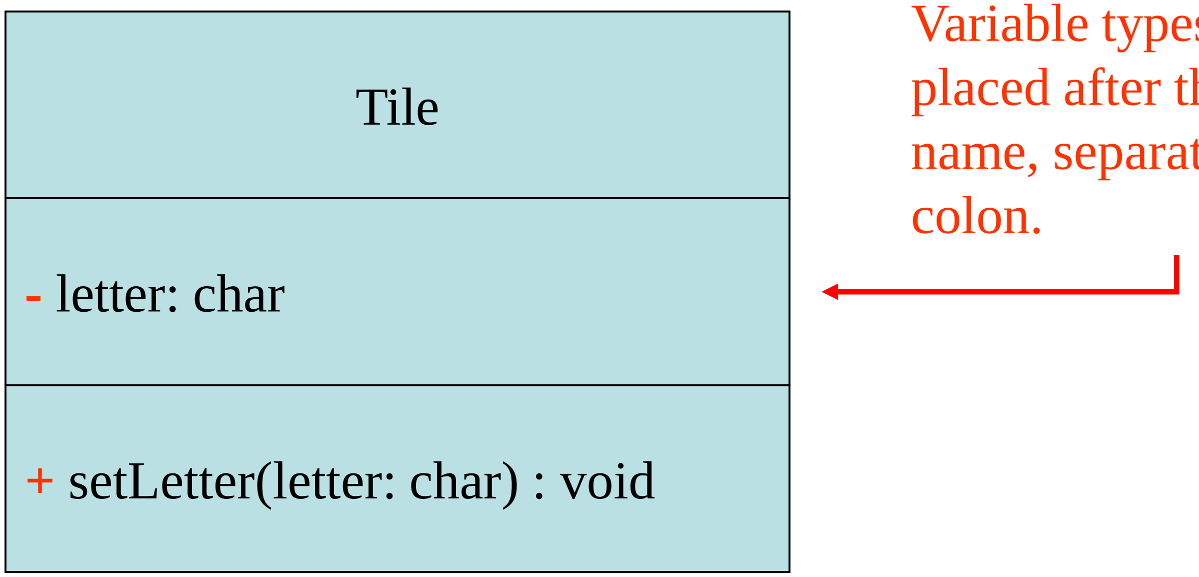
UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.


Access modifiers are denoted as:
+ public
- private
Tile
- letter: char
+ setLetter(letter: char) : void

UML Data Type and Parameter Notation

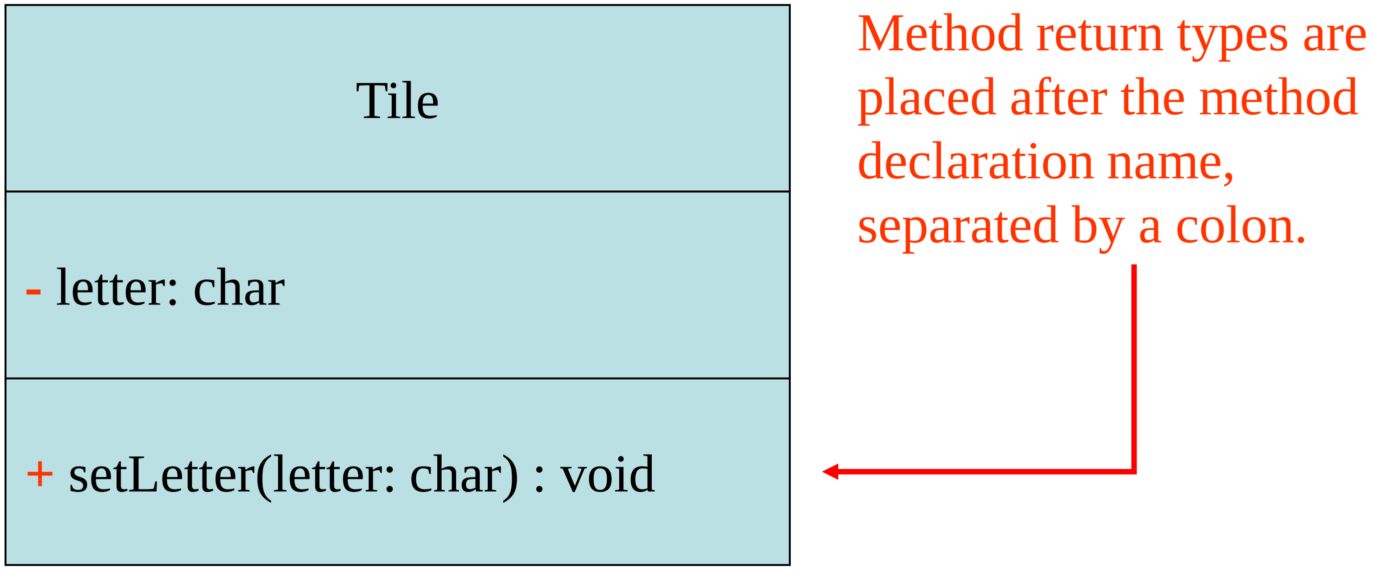
- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



Variable types are placed after the variable name, separated by a colon.

UML Data Type and Parameter Notation

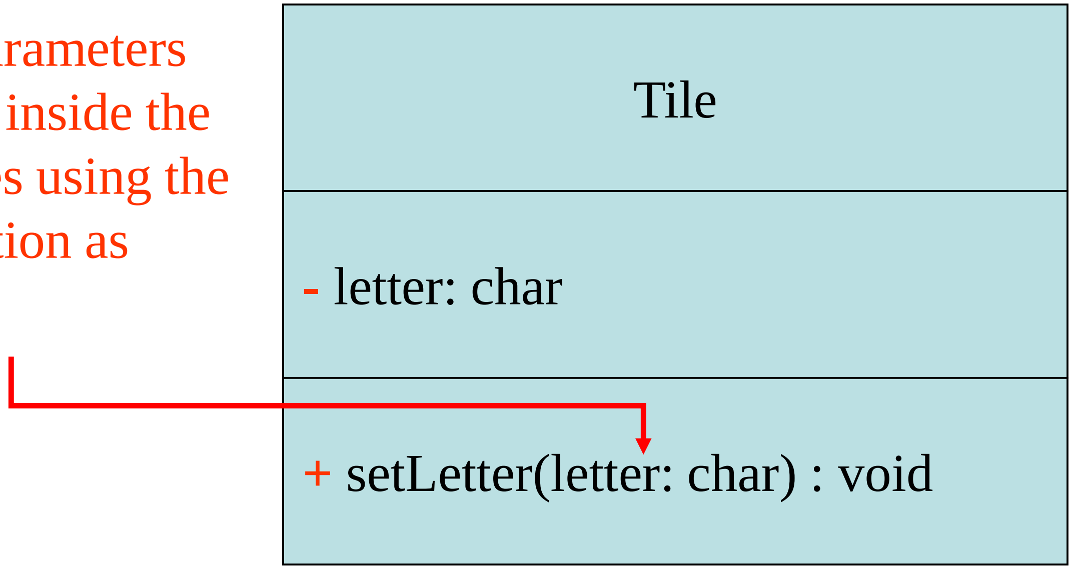
- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.


Tile
- letter: char
+ setLetter(letter: char) : void
Method return types are placed after the method declaration name, separated by a colon.

UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

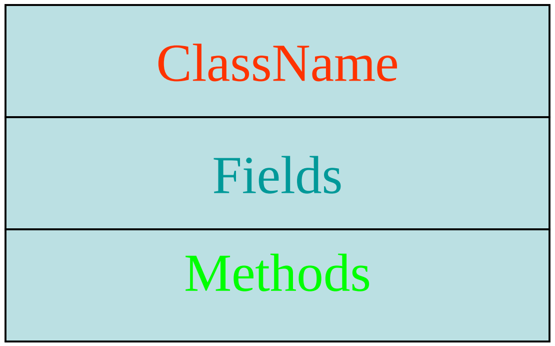
Method parameters are shown inside the parentheses using the same notation as variables.



Converting the UML Diagram to Code

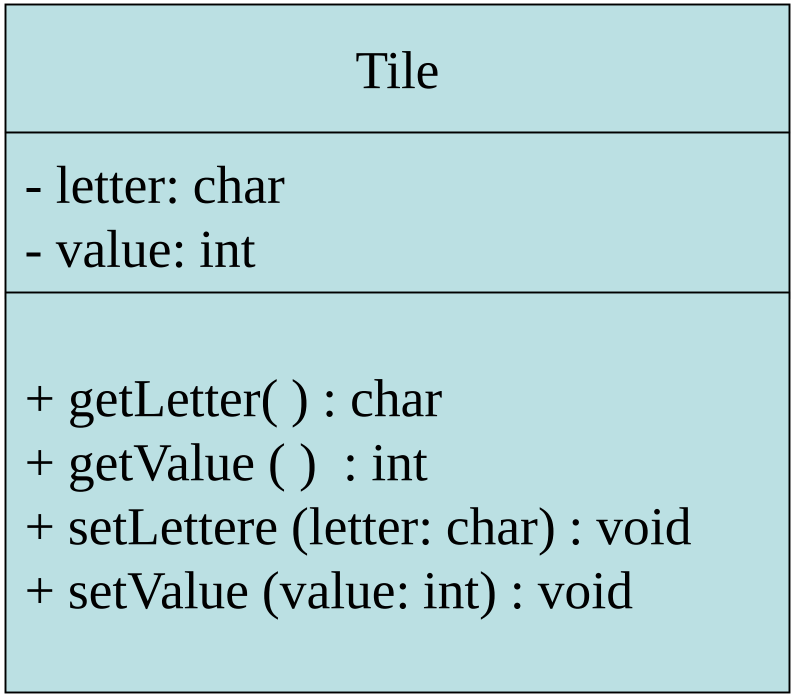
- Putting all of this information together, a Java class file can be built easily using the UML diagram.
- The UML diagram parts match the Java class file structure.

```
class header
{
    Fields
    Methods
}
```



Converting the UML Diagram to Code

The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.



```
public class Tile
{
   private char letter;
   private int value;

   public char getLetter(){
        return this.letter;
   }
   public int getValue(){
        return this.value;
   }
   public void setLetter(char letter){
        this.letter = letter;
   }
   public void setValue(int value){
        this.value = value;
   }
}
```

Constructors

- Constructors have a few special properties that set them apart from normal methods.
  - Constructors have the same name as the class.
  - Constructors have no return type (not even `void`).
  - Constructors may not return any values.
  - Constructors are typically public.

Constructor for `Tile` Class

```
/**
   Constructor
   @param letter The letter of the Tile.
   @param value The value of the Tile.
*/
public Tile(char letter, int value)
{
   this.letter = letter;
   this.value = value;
}
```

The Default Constructor

- When an object is created, its constructor is <u>always</u> called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
  - It sets all of the object’s numeric fields to 0.
  - It sets all of the object’s `boolean` fields to `false`.
  - It sets all of the object’s reference variables to the special value *null*.

The Default Constructor

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The <u>only</u> time that Java provides a default constructor is when you do not write <u>any</u> constructor for a class.
- A default constructor is <u>not</u> provided by Java if a constructor is already written.

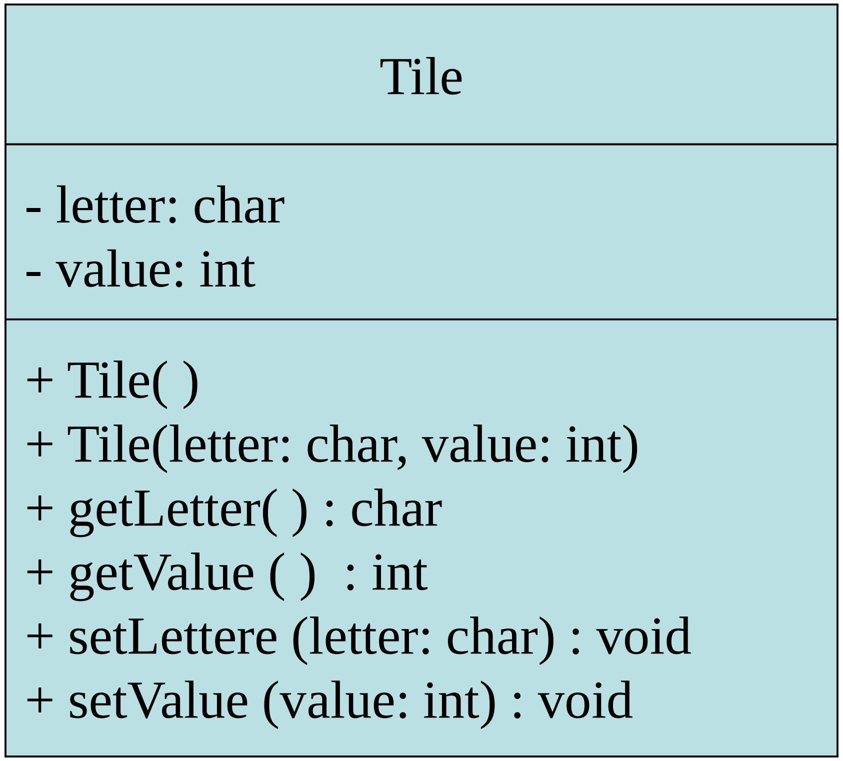
Writing Your Own Default Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg (default) constructor

```
public Tile()
{
     letter = 'A';
     value = 1;
}
```

Constructors in UML

- In UML, the most common way constructors are defined is:

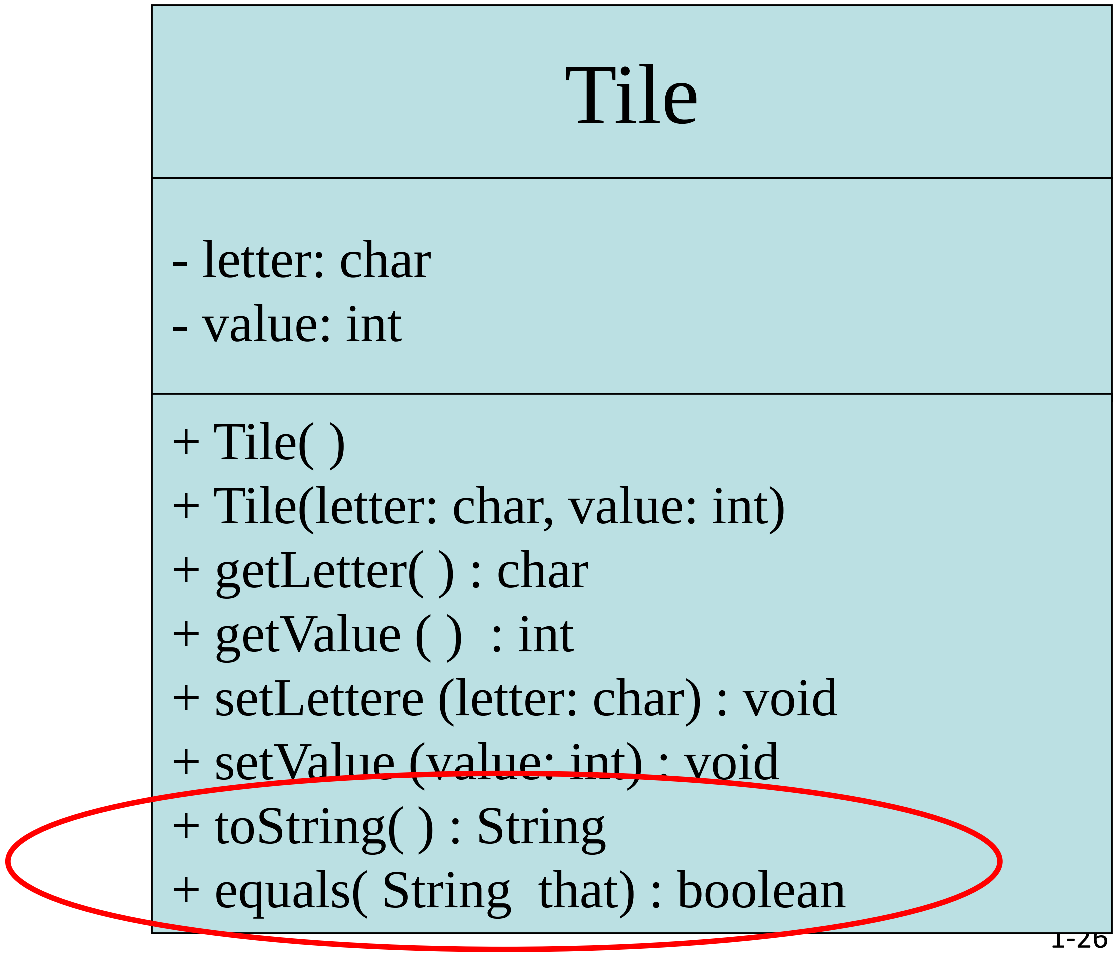


```
Tile
---------------------------------
- letter: char
- value: int
---------------------------------
+ Tile( )
+ Tile(letter: char, value: int)
+ getLetter( ) : char
+ getValue ( )  : int
+ setLettere (letter: char) : void
+ setValue (value: int) : void
```

Notice there is no return type listed for constructors.

The `toString` and `equals` methods

- make the class easier to use


Tile
- letter: char
- value: int
+ Tile( )
+ Tile(letter: char, value: int)
+ getLetter( ) : char
+ getValue ( ) : int
+ setLettere (letter: char) : void
+ setValue (value: int) : void
+ toString( ) : String
+ equals(String that) : boolean

The `toString` method for `Tile` objects

- Returns a String representation of data in object
- invoked automatically whenever object is printed

```
public String toString()
{
   return letter + "/" + value;
}
```

The `equals` method for `Tile` objects

- Returns true if two `Tile` objects have same data

```
public boolean equals(Tile that)
{
   return this.letter == that.letter
          && this.value == that.value;
}
```

Javadoc comments

- New style -- Start with /**, end with */
  - Allows compiler to generate official documentation
  - Keyword `@param` indicates special formatting

```
/**
   Constructor
   @param letter The letter of the Tile.
   @param value The value of the Tile.
*/
public Tile(char letter, int value)
{
   this.letter = letter;
   this.value = value;
}
```

Javadoc comments

- Software developers comment each method for clarity
- Javadoc output can be published straight to web
- Keyword @return indicates special formatting

```
/**
   The getLetter method returns a Tile
   object's letter field.
   @return The value in the letter field.
*/

public char getLetter()
{
   return letter;
}
```

Consider Bug class from GridWorld

- Source Code
- Javadoc generated from source code above
- Javadoc for all GridWorld classes
- Javadoc is a powerful tool for making existing classes easier to use

Agenda

- Review of objects and classes
  - standard class pattern
  - UML diagrams
  - toString, equals, compareTo
  - Javadoc documentation
- OOP's Big 3 Concepts: 🡸
  - Encapsulation
  - Inheritance
  - Polymorphism

Characteristics of OOP

- Objects represent entities in the real world
  - An employee at a company
  - A zombie in a video game
  - A Bug/Rock/Flower in GridWorld
- Majority of methods are ***object methods***
  - like String methods: String s = "hi"; **s**.length();
  - or Tile methods: Tile t = new Tile('Q',8); **t**.setLetter('J');
- Rather than ***class methods***
  - like Math methods: **Math**.sqrt(5); **Math**.random().\

Two Principles of OOP

- **Encapsulation**: objects are isolated from each other by limiting the ways they interact, especially by preventing them from accessing instance variables without invoking methods.
- **Inheritance:** Classes are organized in family trees where new classes extend existing classes, adding new methods and replacing others.
  - OOP design principles lead to cost savings over procedural only design

Encapsulation

- To preserve sanity in large projects
  - limit access that other pieces of code have to our particular object.
  - Stop "silly" (or malicious) programmers from changing the values of our instance variables.
  - For example, in another class file someone writes:

```
Time t = new Time(12,25,35);
t.hour = -15;
System.out.println("Hour is " + t.hour);
```

← forbidden when field marked "private"

Accessing private fields of an object

- Accessor method for Time class, hour field:
  - return the value of an instance variable

```
public int getHour() {
  return hour;
}
```

- Invoking the method:

```
Time t = new Time(10,40,30);
System.out.println("Hour is " + t.getHour());
```

Modifying private fields of an object

- Modifier method for Time class, hour field:
  - change the value of an instance variable

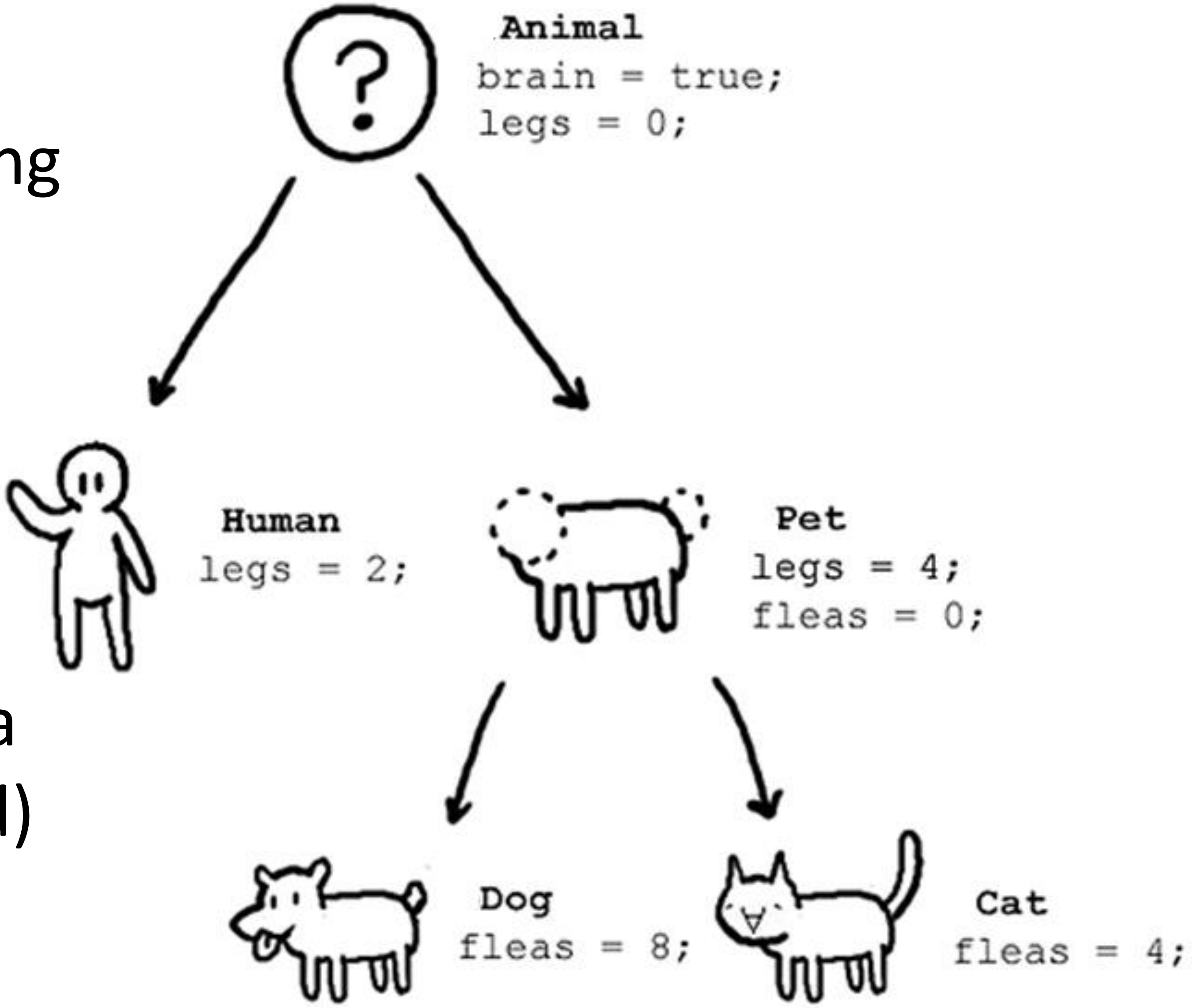
```
public void setHour(int newHour) {
   if (newHour>=0 && newHour < 24)
            hour = newHour;
}
```

- Invoking the method:

```
Time t = new Time(10,40,30);
t.setHour(4);          t.setHour(-15);
```

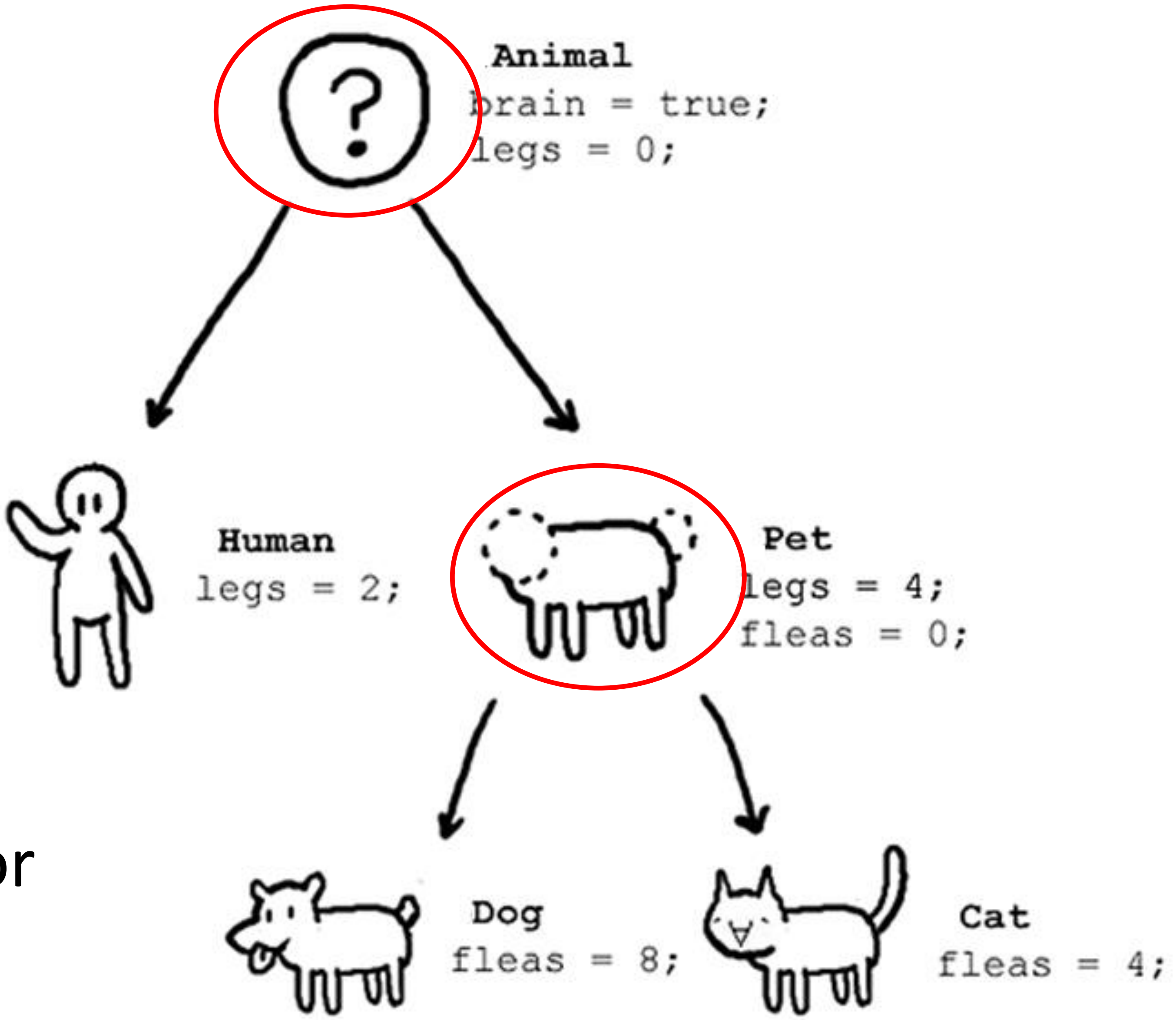
Inheritance

- Saves rewriting code
- Code written for a parent (super) class can be inherited by a child (derived) class



Inheritance

- Some classes are never meant to be instantiated
- They act as prototypes for their child classes to complete



Inheritance Example

```
public class Animal
{
   private boolean brain;
   private int legs;

   public Animal()
   {
   brain = true;
   legs = 0;
   }
```

```
public class Pet extends Animal
{
   private int fleas;

   public Pet()
   {
     super();   // Animal cnstrtr
     setLegs(4);  //Animal mthd
     fleas = 0;    //Pet field
   }
```

Inheritance relationships

- Inheritance should only be used when an "is-a" relationship exists between parent/child
  - a Human "is a"?? Animal **yes**
  - a Dog "is a"?? Pet **yes**
  - An Appointment "is a"?? Time **no!**
- If a "has a" relationship is more suitable
  - use Composition

an Appointment "has a" Time → Composition

Composition Example

```
public class Appointment{
    String where;    // An Appointment "has a"
    Time when;       // location and time

    public Appointment(){     // no arg const
        where = "---";
        when = new Time(0,0,0);
    }
}
```

Inheritance and arrays

- Arrays can only hold objects of the same class
- But inheritance creates "is a" relationships between classes.
- Therefore, the following is legal:

```
Animal [] zoo = new Animal[3];
zoo[0] = new Human();
zoo[1] = new Cat();
zoo[2] = new Dog();
```

- An example of "Polymorphism" – many forms

Start Assignment 15

- OOP is a very deep topic, this has been just a short overview of something we will explore further in CSIS10B.