Data Members

- The fields are the states of the object.
- The field may have an access modifier, say public and private.
  - public: accessible by all classes
  - private: accessible only within its own class
- You can decide if these fields are accessible!
- In practice, all fields should be declared private.
- However, this private modifier does not quarantine any security.\(^1\)
  - What private is good for maintainability and modularity.\(^2\)

---

\(^1\)Thanks to a lively discussion on January 23, 2017.

\(^2\)Read http://stackoverflow.com/questions/9201603/are-private-members-really-more-secure-in-java.
Function Members

• As said, the fields are hidden.
• So we may need getters and setters if necessary:
  • getters: return the state of the object
  • setter: set the state of the object
• For example, \texttt{getX()} and \texttt{getY()} are getters while \texttt{setX()} and \texttt{setY()} are setters in the class \texttt{Point}. 
class Point {
    // data members: fields or attributes
    private double x;
    private double y;

    // function members: methods
    double getX() { return x; }
    double getY() { return y; }

    void setX(double new_x) { x = new_x; }
    void setY(double new_y) { y = new_y; }
}
Exercise: Phonebook

class Contact {
    private String name;
    private String phoneNumber;

    double getName() { return name; }
    double getPhoneNumber() { return phoneNumber; }

    void setName(String new_name) { name = new_name; }
    void setPhoneNumber(String new_phnNum) {
        phoneNumber = new_phnNum;
    }
}

public class PhonebookDemo {

    public static void main(String[] args) {
        Contact c1 = new Contact();
        c1.setName("Arthur");
        c1.setPhoneNumber("09xxnnnnnnn");

        Contact c2 = new Contact();
        c1.setName("Emma");
        c1.setPhoneNumber("09xxnnnnnnn");

        Contact[] phonebook = {c1, c2};

        for (Contact c: phonebook) {
            System.out.printf("%s: %s\n", c.getName(),
                              c.getPhoneNumber());
        }
    }
}
Unified Modeling Language

• Unified Modeling Language (UML) is a tool for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
• Free software:
  • http://staruml.io/ (available for all platforms)

---

Example: Class Diagram for Point

- **Modifiers** can be placed before both fields and methods:
  - `+` for **public**
  - `−` for **private**
A constructor is called by the new operator.
A constructor acts like other methods.
However, its names should be identical to the name of the class and it has no return type.
A class may have several constructors if needed.
Constructors can be overloaded.
Note that the constructors are used only during the objection creation.
Constructors cannot be invoked by any object.
If you don’t define any explicit constructor, Java assumes a default constructor for your class.
Moreover, adding any explicit constructor disables the default constructor.
Parameterized Constructors

- You can provide specific information to the parameterized constructor during the object creation.
- For example,

```java
class Point {
    ...
    // default constructor
    Point() {
        // do something in common
    }
    
    // parameterized constructor
    Point(double new_x, double new_y) {
        x = new_x;
        y = new_y;
    }
    ...
}
```
Self-reference

- You can refer to any (instance) member of the current object within methods and constructors by using this.
- The most common reason for using the this keyword is because a field is shadowed by method parameters.
- You can also use this to call another constructor in the same class by invoking this().
Example: Point (Revisited)

```java
class Point {
    ...
    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

- Note that the `this` operator cannot be used in `static` methods.
Instance Members

- You may notice that, until now, all members are declared w/o static.
- These members are called instance members.
- These instance members are available only after the object is created.
- This implies that each object has its own states and does some actions.
an object reference

ptr into heap

ptr to class data
instance data
instance data
instance data

the heap

the method area

class data
Static Members

- The static members belong to the class\(^4\), and are shared between the instance objects.
- These members are ready once the class is loaded.
  - For example, the main method.
- They can be invoked directly by the class name \texttt{without} using any instance.
  - For example, \texttt{Math.random()} and \texttt{Math.PI}.
- They are particularly useful for utility methods that perform work that is independent of instances.
  - For example, factory methods in design patterns.\(^5\)

\(^4\)As known as class members.

\(^5\)“Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.” by Wikipedia.
Memory used by JVM

<table>
<thead>
<tr>
<th>Area</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Objects</td>
</tr>
<tr>
<td>Stack</td>
<td>- Methods</td>
</tr>
<tr>
<td></td>
<td>- Local Variables</td>
</tr>
<tr>
<td></td>
<td>- Reference Variables</td>
</tr>
<tr>
<td>Code</td>
<td>Byte Code</td>
</tr>
<tr>
<td>Static</td>
<td>Static Methods and Data</td>
</tr>
</tbody>
</table>

-Java Stack

-Perm Gen

-JVM Max Heap Size

-Initial Heap Size

-JVM Memory

-JVM Max Heap (-Xmx value) + JVM Perm Size (-XX:MaxPermSize) + NumberOfConcurrentThreads * (-Xss value) + “other mem”

-Xmx : The JVM max Heap Size

-Xss : The Java Thread Stack Size, the default is OS and JVM dependent, and it can range 256k-to-1MB. The default should be tuned down to a range that doesn’t cause StackOverflow. I often use 128k-192k. Since the default –Xss is high, tuning it down can help save on memory used and given back to the Guest OS.

Perm Size is an area additional to the –Xmx (Max Heap) value and is not GC’ed because it contains class-level information.

“other mem” is additional mem required for NIO buffers, JIT code cache, classloaders, Socket Buffers (receive/send), JNI, GC internal info
• A static method can access other static members. (Trivial.)
• However, static methods cannot access to instance members directly. (Why?)
• For example,

```java
... double getDistanceFrom(Point that) {
    return Math.sqrt(Math.pow(this.x - that.x, 2) + Math.pow(this.y - that.y, 2));
}

static double measureDistanceBetween(Point p1, Point p2) {
    // You cannot access to x and y directly!
    return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
}
... 
```
Example: Count of Points

class Point {
    ...
    private static int numOfPoints = 0;

    Point() {
        numOfPoints++;
    }

    Point(int x, int y) {
        this(); // calling the constructor with no input argument; should be placed in the first line in the constructor
        this.x = x;
        this.y = y;
    }
    ...
}
Exercise: Singleton

- In some situations, you may create the only instance of the class.

class Singleton {
    // Will be ready as soon as the class is loaded.
    private static Singleton INSTANCE = new Singleton();

    // Do now allow to invoke the constructor by other classes.
    private Singleton() {}

    // Only way to obtain this singleton by the outside world.
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
Garbage Collection (GC)\(^6\)

- Java handles deallocation **automatically**.
  - Heuristics: period or memory stress.
- GC is the process of looking at the **heap** memory, identifying whether or not the objects are in use, and deleting the unreferenced objects.
- An object is said to be **unreferenced** if the object is no longer referenced by any part of your program.
  - Simply assign **null** to the reference to make the object unreferenced.
- So the memory used by these objects can be reclaimed.

\(^6\)http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
• The method `finalize()` conducts a specific task that will be executed right before the object is reclaimed by GC.
  • For example, closing files and terminating network connections.
• The `finalize()` method can be only invoked prior to GC.
• In practice, it must not rely on the `finalize()` method for normal operations. (Why?)
```java
public class Garbage {
    private static int numOfObjKilled = 0;

    public void finalize() {
        numOfObjKilled++;
    }

    public static void main(String[] args) {
        double n = 1e7;
        for (int i = 1; i <= n; i++)
            new Garbage(); // lots of unreferenced objects
        System.out.println(numOfObjKilled);
    }
}
```

- You may try different number for instance creation.
- The number of the objects reclaimed by GC is uncertain.
HAS-A Relationship

- **Association** is a weak relationship where all objects have their own lifetime and there is no ownership.
  - For example, teacher $\leftrightarrow$ student; doctor $\leftrightarrow$ patient.
- If A uses B, then it is an **aggregation**, stating that B exists independently from A.
  - For example, knight $\leftrightarrow$ sword; company $\leftrightarrow$ employee.
- If A owns B, then it is a **composition**, meaning that B has no meaning or purpose in the system without A.
  - For example, house $\leftrightarrow$ room.
Example: Lines

- +2: two **Point** objects used in one **Line** object.
More Examples

- Circle, Triangle, and Polygon.
- Book with Authors.
- Lecturer and Students in the classroom.
- Zoo with many creatures, say Dog, Cat, and Bird.
- Channels played on TV.
- More.
More About Objects

- **Inheritance**: passing down states and behaviors from the parents to their children.
- **Interfaces**: requiring objects for the demanding methods which are exposed to the outside world.
- **Polymorphism**
- **Packages**: grouping related types, and providing access controls and name space management.
- **Immutability**
- **Enumeration types**
- **Inner classes**
The relationships among Java classes form **class hierarchy**.

We can define new classes by **inheriting** commonly used states and behaviors from predefined classes.

A class is a **subclass** of some class, which is so-called the **superclass**, by using the `extends` keyword.

- For example, `B extends A`.

In semantics, **B is a special case of A**, or we could say **B specializes A**.

- For example, human and dog are two specific types of animals.

When both **B and C** are subclasses of **A**, we say that **A generalizes B and C**. (Déjà vu.)

Note that Java allows **single inheritance** only.
class Animal {
    String name;
    int weight;
    Animal(String s, int w) { name = s; weight = w; }
    void eat() { weight++; }
    void exercise() { weight--; }
}

class Human extends Animal {
    Human(String s, int w) { super(s, w); }
    void writeCode() {}
}

class Dog extends Animal {
    Dog(String s, int w) { super(s, w); }
    void watchDoor() {}
}
Class Hierarchy\textsuperscript{7}

\textsuperscript{7}See Fig. 3-1 in p. 113 of Evans and Flanagan.
Recall that the keyword \texttt{this} is used to refer to the object itself.

You can use the keyword \texttt{super} to refer to (non-private) members of the superclass.

Note that \texttt{super()} can be used to invoke the constructor of its superclass, just similar to \texttt{this()}. 
Constructor Chaining

• As the constructor is invoked, the constructor of its superclass is invoked accordingly.

• You might think that there will be a whole chain of constructors called, all the way back to the constructor of the class Object, the topmost class in Java.

• So every class is an immediate or a distant subclass of Object.

• Recall that the method finalize() and toString() are inherited from Object.
  • toString(): return a string which can be any information stored in the object.
Example

```java
class A {
    A() { System.out.println("A is creating..."); } 
}

class B extends A {
    B() { System.out.println("B is creating..."); } 
    // overriding toString()
    public String toString() { return "I am B."; } 
}

public class ConstructorChainingDemo {
    public static void main(String[] args) {
        B b = new B();
        System.out.println(b);
    }
}
```

- The `println()` method (and similar methods) can take an object as input, and invoke `toString()` method implicitly.
Method Overriding

- The subclass is allowed to change the behavior inherited from its superclass, if needed.
- If one defines an instance method with its method name, parameters, and also return type, all identical to the previously defined method in its superclass, then we say this newly-defined method overrides the one in the superclass.\(^8\)
  - Recall that method overloading occurs only in the same class.
- Note that you can invoke the overridden method through the use of the keyword `super`.

\(^8\)Notice that the static methods do not follow this rule.
Association of the method definition to the method call is known as **binding**.

The binding which can be resolved at the compilation time is known as **static binding or early binding**.

- They are the **static, private or final** methods.\(^9\)

If the compiler is not able to resolve the binding, such binding is known as **dynamic binding or late binding**.

- For example, method overriding.

\(^{9}\)We will see the **final** keyword soon.
- When there are multiple implementations of the method in the inheritance hierarchy, the one in the “most derived” class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference variable of the superclass type.
  
  - As you can see in Cat Simon.

- This is so-called **subtype polymorphism**.