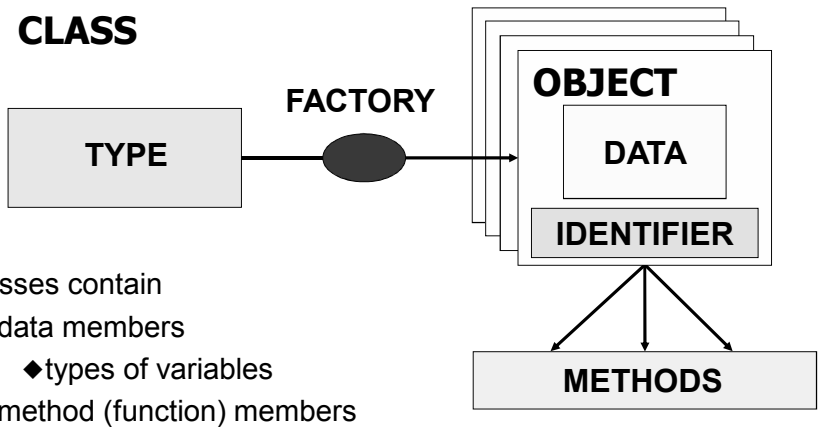
Objects and Classes: Working with the State and Behavior of Objects

The Core Object-Oriented Programming Concepts



- Classes contain
  - ◆data members
    - ◆types of variables
  - ◆method (function) members
    - ◆operations which can be performed on variables

A World of Objects

- Software objects are artifacts abstracting real-life objects
- A set of objects with common characteristics and behaviors is represented as a class:
  - All cars belong to the class of Cars
  - All clocks belong to the class of Clocks
- An OO program (e.g., in Java) is a set of objects interacting with each other
  - Objects asks other objects to fulfill the assigned responsibilities

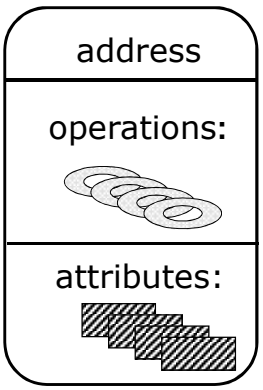
3

What's an object?

Object **=**

- **State** = Data (attribute values) *encapsulated* by the object
- **Behavior** = How object reacts (operations) to *State Changes* & *Messages*
- **Identity** = The address of the object in main memory

Identity **+** ***Behavior*** **+** ***State***



4

Example of a Student Object

- The state of a student include its:
  - name, age, sex, address, grades, …
- The behavior of a student allows to:
  - move from one house to another (the address can be changed),
  - add a new grade to a course (the university record can be updated)
- Why Create Objects?
  - Keeps all related info (i.e., data) together
  - Refer to all the related info by one name
  - Keep methods together with related data
  - Protect the information

Student_Jini

Move(UC)

AddGrade(cs252, 10)

Name: Jini

Age: 20

Sex: T

Object State and Types

- The state of an object is structured:
  - Attributes have names
  - Attribute values have types
- Attribute types may be:
  - Primitive data types
  - References to other objects
- The same is true for operation arguments

string

Int

boolean

Student_Jini

Move(UC)

AddGrade(cs252, 10)

Name: Jini

Age: 20

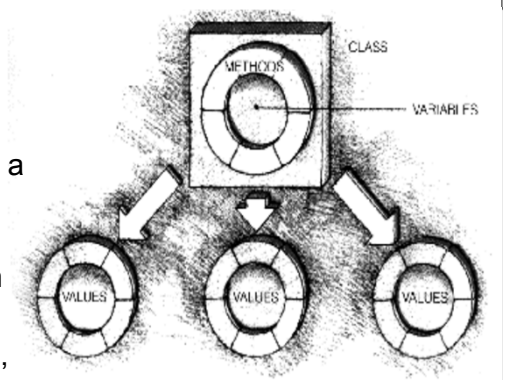
Sex: T

How Objects are Manufactured?

- A class defines object attributes and associated operations by specifying their respective data types and method parameters
  - It is the template or pattern from which objects are produced
- Creating an object from a class is called instantiation
  - All objects instances of a particular class will be of the same type, they will have the same state structure and methods
- The class declaration is a way of defining new data types for your program extending your language basic "vocabulary"
- A class defines the data types for an object, but a class does not store data values
  - Each object has its own unique data space

How Objects are Manufactured?

- Object state is stored in instance variables (or attributes)
  - each instance of Student (each object) has three values to store its state: a string, an integer and a boolean
  - each instance of a class (object) maintains its own set of values in the instance variables
  - If there are 1000 Student objects, there are 1000 names and 1000 ages and 1000 sexes



Copyright: OOT A Managers' perspective, Dr. Taylor

How to Declare Classes?

- The class template for Student objects is declared as follows:

```
class Student {
   public String name;
   public int age;
   public bool sex;
};
```

The "public" keyword makes the instance variables accessible outside of the class

How to Create Objects?

- It can be used to create a single Student object

```
class Student {
   public String name;
   public int age;
   public bool sex;
};
```

```
Student Student_Jini = new Student();
```

This variable has a type "reference to the object of type Student"

The new command is used to construct an object of a specific class

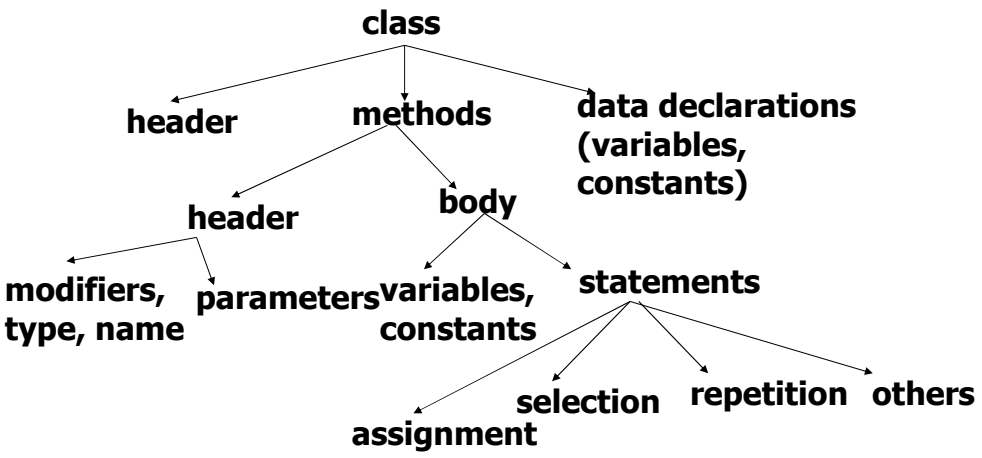
How to Initialize Instance Variables ?

- ...and initialize their instance variables individually

```
Student Student_Jini = new Student();

Student_Jini.name = "My text";
Student_Jini.age = 20;
Student_Jini.sex = T;
```

The instance variables of Student are accessed using the '.' operator

Elements of a Class Definition



Creating Objects

- To create a new object you need to allocate a memory for the object
- The keyword `new` followed by class name and a pair of parenthesis creates an object in Java
  - Objects are always initialized when they are created
  - If creation requires additional information about their initial state it is written inside the parenthesis (parameters)
- A class constructor :
  - is a special method that is used to set up a newly created object
  - often sets the initial values of variables
  - has the same name as the class
  - does not return a value
  - has no return type, not even `void`

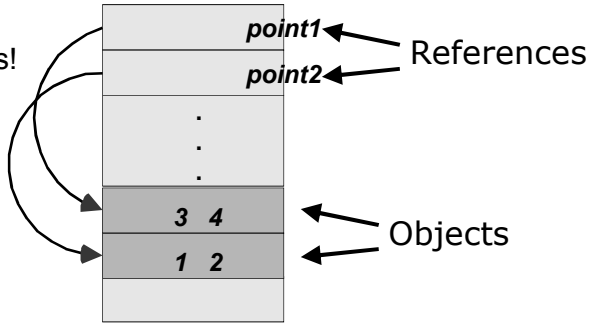
Referencing Objects

- Objects are created in the main memory but have no name and cannot be found
- In order to interact and use them we have to assign a reference to the address of the allocated memory
  - We refer to an object by means of an object reference
  - An object reference holds the data about memory location of the object
- Usually object references are stored in program variables

Example of Object References

```
Point point1 = new Point(3,4);
Point point2 = new Point(1,2);
```

- An object reference and an object are two different things!
- Both are stored in memory but in different places!
- What can we do with references?
  - Define it
  - Assign a new value to it

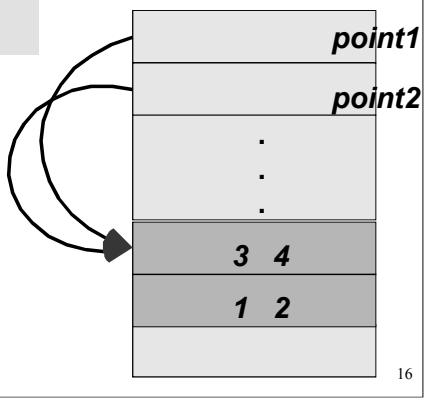


15

Object References Assignment

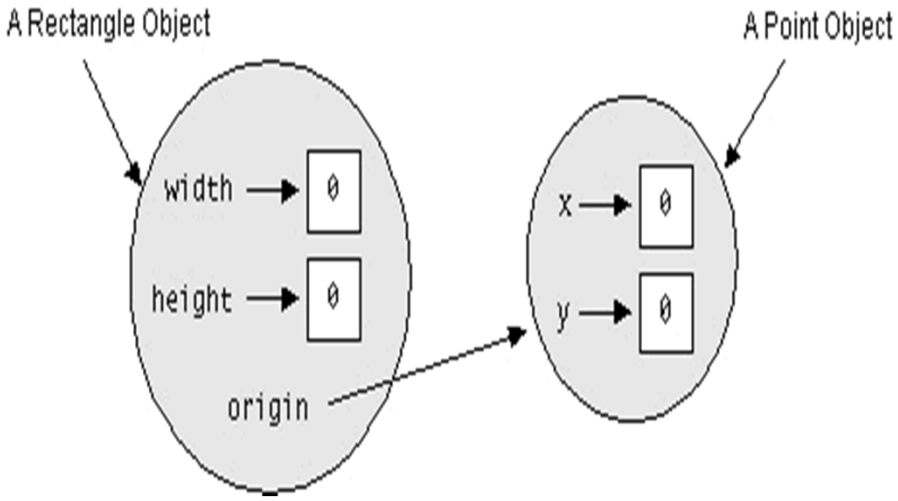
```
Point point1=new Point(3,4);
Point point2=new Point(1,2);
point2 = point1;
int newx = point2.getx();
// newx is 3!
```

- The assignment, copies the value of the reference ***point1*** to ***point2***
- Several references that refer to the same object are called aliases of each other
- Changes that are done to the object through one reference effects all aliases, because they actually refer to the same memory location!



16

Objects and References: Example



17

Memory Allocation

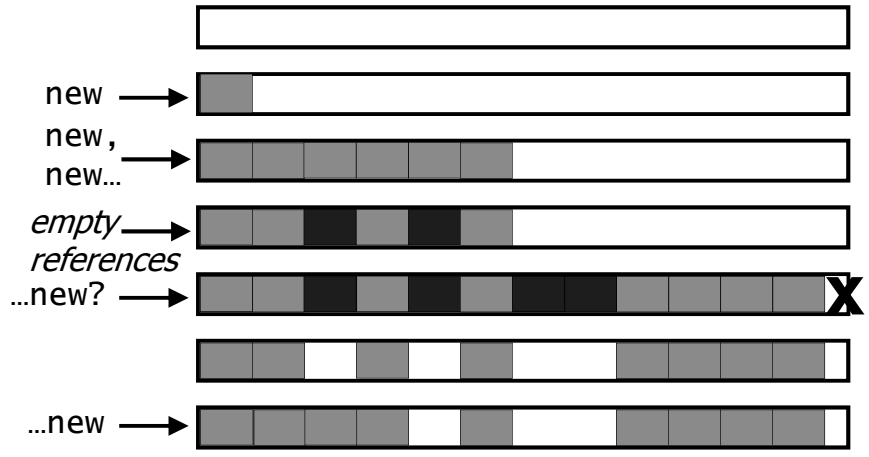
- Why is it important to allocate a specific place in memory for each object?
  - Preserve the object state
  - Refer to it from different places
  - Not destroy it by overriding it with other data in that memory cells
- In previous example a reference to the original object `point2` is gone
  - No other references refers to this memory location
- When an object no longer has any valid references to it, it can no longer be accessed by the program
  - It is useless, and therefore called garbage
- In Java, objects are automatically destroyed when they are no longer needed

18

Garbage Collection

- Garbage collection takes care of deallocating the memory storage
- The basic idea is storage reuse:
  - When an object is created the memory space is allocated for this object
  - Later if the object or data are not used the memory is returned to the system for future reuse
- Java performs automatic garbage collection periodically, returning an object's memory to the system for future use
  - In other languages, the programmer has the responsibility for performing garbage collection

Allocation Example

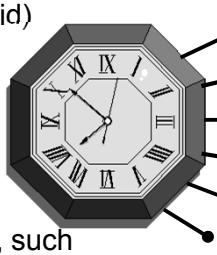


Object Methods

- Object's data is useless unless we can manipulate it
  - We manipulate the data of an object by sending it messages
- For this purpose, we must provide methods that define its behavior
  - Methods are a part of the class definition
  - We send messages to objects by invoking their methods
- Methods are declared in a similar way to the method main. Only there's a big difference:
  - The main method was static, which means it wasn't bound to a specific object
  - We want to declare instance methods, which operate on a specific instance of an object

Example of a Clock Object Methods

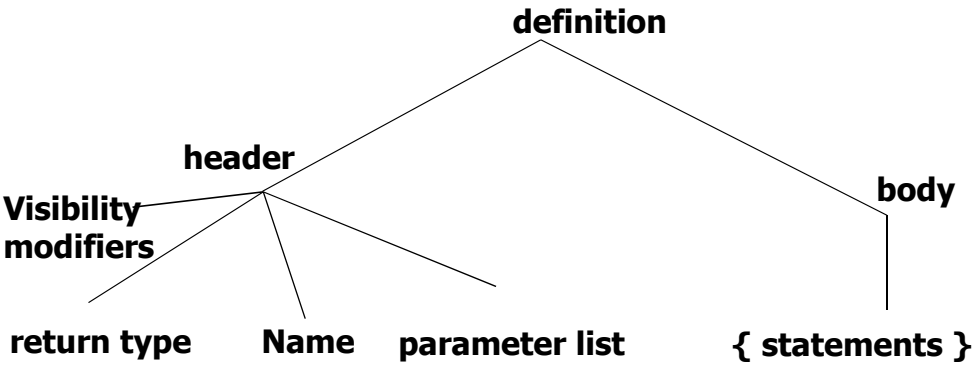
- Methods declaration comprise:
  - visibility modifiers
  - return-type (perhaps void)
  - method-name
  - parameter-list (perhaps void)
  - statement-body



```
Clock(int hours,
      int minutes,
      int seconds)
void hourElapsed()
int getSeconds()
int getMinutes()
int getHours()
...
```

- Why methods?
  - object-oriented languages, such as C++ and Java, use methods to give objects the capability to perform tasks

Elements of a Method Definition



Declaring Instance Methods

```
public class Clock {
  private int hours,minutes,seconds;
  public Clock() {
     // ... }
   /**
    * Advances the clock by one hour
    */
   public void hourElapsed( ) {
      hours = (hours + 1) % 24; }
  /**
   * Returns the hour read
   */
  public int getHours( ) {
     return hours; }
}
```

Declaring Instance Methods

```
public class Clock {
  private int hours,mir
  public Clock() {
     // ... }
   /**
    * Advances the clock by one hour
    */
   public void hourElapsed( ) {
     hours = (hours + 1) % 24; }
  /**
   * Returns the hour read
   */
  public int getHours( ) {
     return hours; }
}
```

The "public" keyword makes the instance methods accessible outside of the class

Declaring Instance Methods

```
public class Clock {
  private int hours,minutes,second
  public Clock() {
     // ... }
   /**
    * Advances the clock by one hour
    */
   public void hourElapsed( ) {
     hours = (hours + 1) % 24; }
  /**
   * Returns the hour read
   */
  public int getHours( ) {
     return hours; }
}
```

The name of Instance methods

Declaring Instance Methods

```
public class Clock {
  private int hours,minutes,seconds;
  public Clock() {
     // ... }
   /**
    * Advances the clock by one
    */
   public void hourElapsed( ) {
      hours = (hours + 1) % 24; }
  /**
   * Returns the hour read
   */
  public int getHours( ) {
     return hours; }
}
```

The parameters of instance methods

Declaring Instance Methods

```
public class Clock {
  private int hours,minutes,seconds;
  public Clock() {
     // ... }
   /**
    * Advances the clock by one
    */
   public void hourElapsed( ) {
      hours = (hours + 1) % 24; }
  /**
   * Returns the hour read
   */
  public int getHours( ) {
     return hours; }
}
```

The body of instance methods

Declaring Instance Methods

```
public class Clock {
  private int hours,minutes,se
  public Clock() {
     // ... }
   /**
    * Advances the clock by
    */
   public void hourElapsed() {
      hours = (hours + 1) % 24; }
  /**
   * Returns the hour read
   */
  public int getHours() {
     return hours; }
}
```

The "void" keyword denotes that the method has no return value

Declaring Instance Methods

```
public class Clock {
  private int hours,minutes,seconds;
  public Clock() {
     // ... }
   /**
    * Advances the clock by one hour
    */
   public void hourElapsed() {
      hours = (hours + 1) % 2
  /**
   * Returns the hour re
   */
  public int getHours() {
     return hours; }
}
```

The "return" keyword specifies the actual value to be returned

Methods Return Types and Statement

- The return type of a method indicates the type of value that the method sends back to the calling client
  - The return-type of **`getHours()`** is **`int`**. When a client asks for the hours read of a clock it gets the answer as an `int` value
  - A method that does not return a value (e.g., **`hourElapsed()`**) has a **`void`** return type

- In the method definition, the **`return`** statement specifies the value of the **`<expression>`** that should be returned, which must conform with the return type of the method
  - If the method returns **`void`** you can simply use **`return`**
  - or you can omit the statement

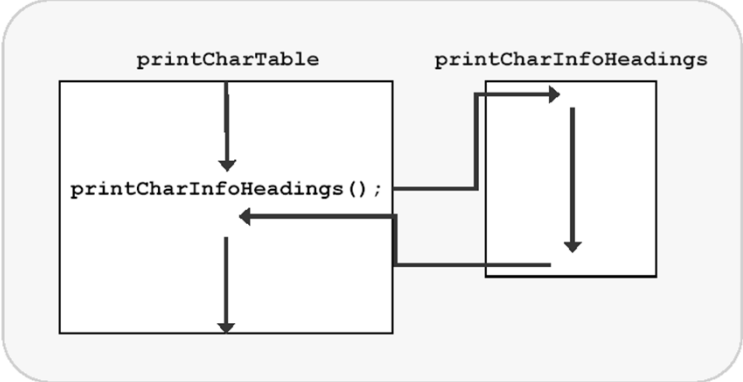
Using Instance Methods

```
public class ClockTest {
  public static void main(String[] args) {
    Clock swatch = new Clock();
    Clock seiko = new Clock();
    System.out.println(swatch.getHours()); // 0
    System.out.println(seiko.getHours());  // 0
    swatch.hourElapsed();
    System.out.println(swatch.getHours()); // 1
    System.out.println(seiko.getHours());  // 0
  }
}
```

The instance methods of Clock are accessed using the '.' operator

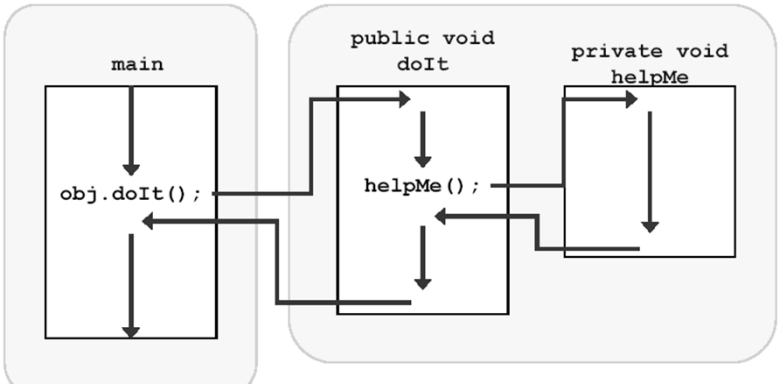
Execution Flows from one Method to Another

- The called method can be within the same class, in which case only the method name is needed



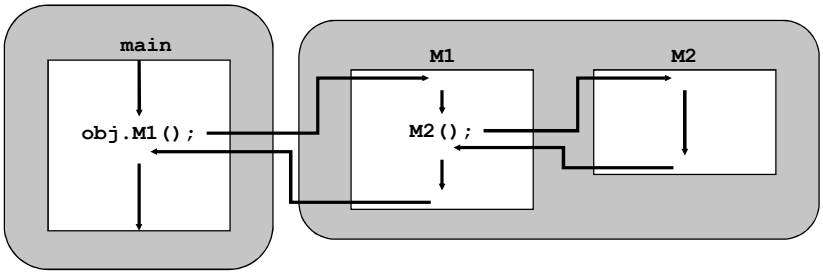
Control also Flows between Classes and Back

- The called method can be part of another class or object



Method Control Flow

- Instead of writing a long list of instructions we write a collection of methods and invoke (or call) them one after another
- We decomposed a problem into series of simpler methods
  - The **`main`** method is invoked by the system when you submit the bytecode to the interpreter
  - The invoked method could be part of another class or object
  - Each method call returns to the place that called it



Method Parameters and Signatures

- A method can be defined to accept zero or more parameters
- Each parameter in the parameter list is defined by its type and name
  - The parameters in the method definition are called formal parameters
  - The values passed to a method when it is invoked are called arguments, or actual parameters
- The name of the method together with the list of its formal parameters is called the signature of the method

```
public void setTime(int hours, int minutes,  int
  seconds)

"void_setTime_int_int_int"
```

SetTime Method Documentation

```
public class Clock {
  private int hours, minutes, seconds;
  /**
   * Sets the clock to the specified time.
   * If one of the parameters is not in the allowed
   * range, the call does not have any effect on the
   * clock.
   * @param hours The hours to be set (0-23)
   * @param minutes The minutes to be set (0-59)
   * @param seconds The seconds to be set (0-59)
   */
```


SetTime Method Implementation

```
public class Clock {
  private int hours, minutes, seconds;
  /**  … */

public void setTime(int hours, int minutes, int
  seconds) {
       if ((seconds >= 0) && (seconds < 60) &&
           (minutes >= 0) && (minutes < 60) &&
           (hours >= 0) && (hours < 24)) {
              this.hours = hours;
              this.minutes = minutes;
              this.seconds = seconds;
       }
       // no effect if input is illegal
   }
}
```

SetTime Method Implementation

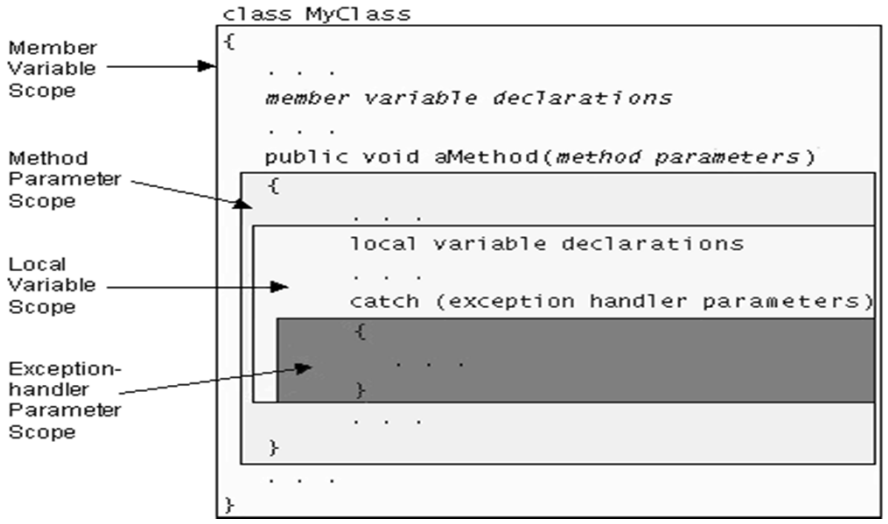
```
public class Clock {
  private int hours, minut
  /**  … */

public void setTime(int ho
  seconds) {
       if ((seconds >= 0) && (seconds < 60) &&
           (minutes >= 0) && (minutes < 60) &&
           (hours >= 0) && (hours < 24)) {
              this.hours = hours;
              this.minutes = minutes;
              this.seconds = seconds;
       }
       // no effect if input is illegal
   }
}
```

The "this" keyword denotes a reference to the object that the method is acting upon

Scope and Lifetime of Variables

- The scope of an identifier declaration is t*he region of text* in which the declaration is effective
  - E.g., class scope, block scope, global, etc.
- The lifetime of a variable (or an object) bound to an identifier is the period during which it exists in the memory
  - E.g., Static variables and objects allocated via new on *heap* have "infinite" lifetime (garbage collection)
  - E.g., Formal parameters and local variables of a method are allocated on. *stack* (LIFO discipline)

Blocks and Scopes



Local vs. Global Data

- A block is the set of statements (a *compound statement)* between a pair of matching braces (curly brackets)
  - A variable declared inside a block is known only inside that block
- *Local data* is accessible only by the method in which they reside
  - Local data are created when a method is invoked and they are destroyed when a method completes its execution
  - The garbage collector destroys local data automatically
- *Global data* are accessible by all sections of your program on an equal basis (Java does not have them in the proper sense of the term)
  - any variable declared outside of all methods is global within the class
  - any variable declared within a method is local to that method unless explicitly made public

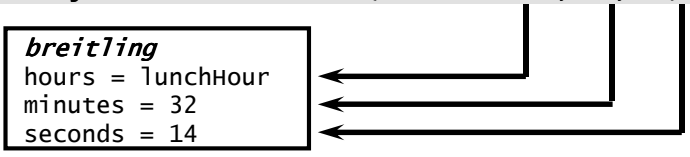
When and Where to Declare Variables

- Variables in Java can be declared only once for a method block
  - although the variable does not exist outside the block, other blocks in the same method cannot reuse the variable's name
- Good programming practice:
  - declare variables just before you use them
  - initialize variables when you declare them
  - do not declare variables inside loops
    - it takes time during execution to create and destroy variables, so it is better to do it just once for loops)
  - it is ok to declare loop counters in the initialization field of **`for`** loops, e.g.
    `for(int i=0; i <10; i++)…`
    - the initialization field executes only once, when the **`for`** loop is first entered

Passing Parameters during Method Invocation

- Both primitive types and object references can be passed as parameters
  - When a primitive type is passed, a copy of the actual value is made and assigned to the formal parameter(“call by value”)
  - When an object reference is passed, the formal parameter becomes an *alias* of the actual parameter (“call by reference”)

```
Clock my_clock = new Clock();
int lunchHour = 12;
my_clock.setTime(lunchHour,32,14);
```



Parameters and Local Variables

- If the argument in a method invocation is a variable, then it is the value of the variable that is plugged in, not the variable name
- The number and type of arguments must be the same as the number and type (up to sub-typing) of formal parameters and in the same order
  - Formal parameters are local to their method
- Java will perform an automatic type conversion (type cast) if you use an argument in a method call that does not match the type of the formal parameter
  - If the type of the argument in a method call is int and the type of the formal parameter is double, then Java will convert the value of type int to the corresponding value of type double

More on Class Constructors

- A class can have many ways to initialize its instances
  - This is done by defining several constructors
- Different constructors differ by the number and/or type of parameters they receive
  - When we construct an object, the compiler decides which constructor to invoke according to the type of the actual parameters
  - If no other constructor is defined, the compiler creates the default constructor for the class automatically

Default and Overloaded Clock Constructors

```
/**
 * Constructs a new clock with the specified hours,
 * minutes and seconds read.
 * If one of the paramenters is not in the allowed
 * range, the time will be reset to 00:00:00.
 * @param hours The hours to be set (0-23)
 * @param minutes The minutes to be set (0-59)
 * @param seconds The seconds to be set (0-59)
 */
 public class Clock {
    private int hours, minutes, seconds;
    public Clock() {
        // initializes all the instance variables to
        // zero or null (depending on the type)
      }
    public Clock(int hours, int minutes, int seconds) {
       // implementation on the next slide
 } }
```


Clock Constructor Implementation

```
public Clock(int hours, int minutes, int seconds) {
   if ((seconds >= 0) && (seconds < 60) &&
       (minutes >= 0) && (minutes < 60) &&
       (hours >= 0) && (hours < 24)) {
          this.hours = hours;
          this.minutes = minutes;
          this.seconds = seconds;
   }
   else {
      this.hours = 0;
      this.minutes = 0;
      this.seconds = 0;
   }
}
 Clock c1 = new Clock();

 Clock c2 = new Clock(23,12,50);
```

Clock Constructor Implementation

```
public Clock(int hours, int minutes, int seconds) {
   if ((seconds >= 0) && (seconds < 60) &&
       (minutes >= 0) && (minutes < 60) &&
       (hours >= 0) && (hours < 24)) {
          this.hours = hours;
          this.minutes = minutes;
          this.seconds = seconds;
   }
   else {
      this.hours = 0;
      this.minutes = 0;
      this.seconds = 0;
   }
}
 Clock c1 = new Clock();
 Clock c2 = new Clock(23,12,50);
```

The default constructor is used

Clock Constructor Implementation

```
public Clock(int hours, int minutes, int seconds) {
   if ((seconds >= 0) && (seconds < 60) &&
       (minutes >= 0) && (minutes < 60) &&
       (hours >= 0) && (hours < 24)) {
          this.hours = hours;
          this.minutes = minutes;
          this.seconds = seconds;
   }
   else {
      this.hours = 0;
      this.minutes = 0;
      this.seconds = 0;
   }
}
 Clock c1 = new Clock();
 Clock c2 = new Clock(23,12,50);
```

The overloaded constructor is used

Overloading Methods

- Non-constructor methods can also be overloaded!
  - The overloaded functions have the same name but must differ in the number or type of parameters
  - They all must return the same type!
- Advantages:
  - Opens up namespace
  - Means shorter names
  - An overloaded constructor provides multiple ways to set up a new object
- Disadvantages:
  - Can accidentally invoke wrong version as a result of human error (leaving off parameter, forgetting a cast)

Contract and Implementation of a Class

- *Contract* of a class is defined by a set of methods (and sometimes publicly available attributes) and their associated semantics
- *Implementation* of a class is defined by the implementation of the methods the class supports

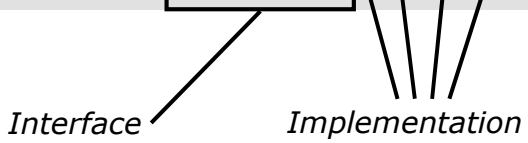
```
class Student {
  private String name;
  Student(String name) {
     this.name = name;
  }
  public String getName( ) { return name; }
}
```

contract **implementation**

Object Interface vs. Implementation

- Objects have two kinds of behavior:
  - Outer – send & receive messages, relations with other objects
  - Inner – processing and computing the messages
- Usually these are called the interface (outer) vs. Implementation (inner)

```
class Point {
  float m_x,m_y;
  public float getx()   { return m_x; }
  public float gety()   { return m_y; }
  public void setx(float x) { m_x=x; }
  public void sety(float y) { m_y=y; }
}
```

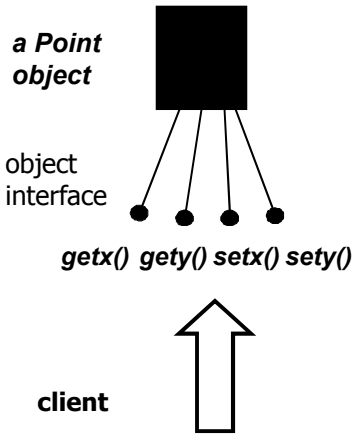


Encapsulation

- An object should be self-governing; any changes to the object's state (its variables) should be accomplished by that object's methods
  - We should make it difficult, if not impossible, for another object to "reach in" and alter an object's state
  - The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished
- This makes the use of objects more easy and allows later changes in the implementation

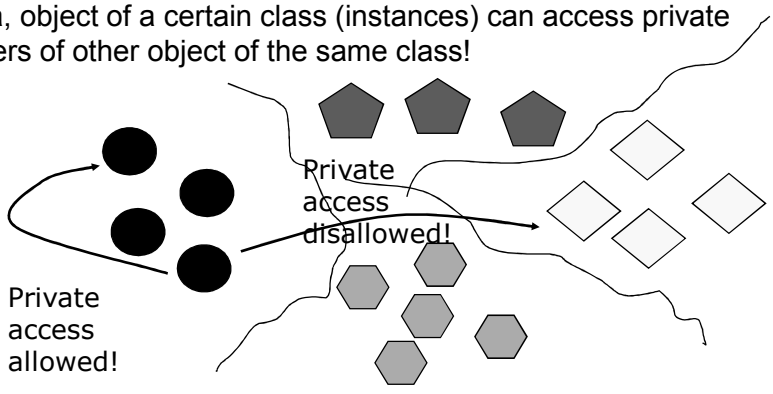
Encapsulation

- An encapsulated object can be thought of as a black box
  - its inner workings are hidden from other objects, clients or users
- Why Encapsulation?
  - In black box programming user gets object interface but should not have to be aware of internal implementation of it
  - The interface cannot be bypassed
  - Encapsulation makes an object easy to manage because all communication with the object are done through well defined services



Encapsulation Level

- Encapsulation comes in the class level and not in the object level
- Sometimes object instances of the same class need to access each other's "guts" e.g., for state copying
  - if we want to create an identical instance of an existing object
- In Java, object of a certain class (instances) can access private members of other object of the same class!



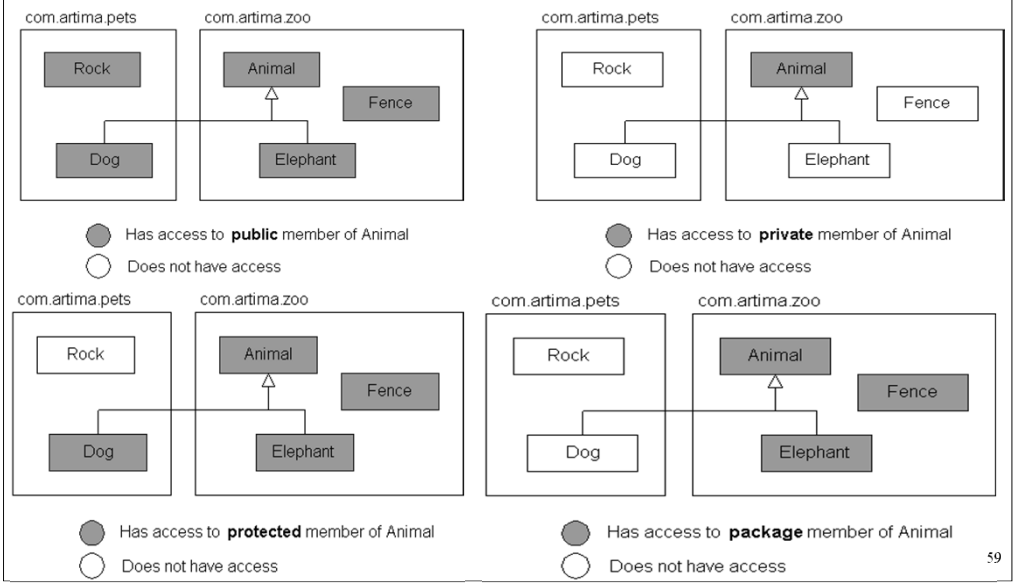
Object Protection and Visibility Modifiers

- Encapsulation is a powerful abstraction
  - An abstraction hides the right details at the right time
- We accomplish encapsulation through the use of appropriate visibility modifiers
  - Minimize the chance that one section of your program will improperly change objects intended for other sections
- Why protect the members of our objects?
  - Sometimes data elements depend on each other
  - Only some variables in a class belong to the “public” interface
  - Bugs can be isolated: Public member data could be altered from anywhere

Java Class Visibility Modifiers

- Java has *four* visibility modifiers specifying respective levels of protected access to class objects
  - Public – permits everyone to access the members of a class
    - accessed by all classes in all packages
  - Private – hides completely the class's members from outside classes as well as from subclasses
    - accessed only within the declaring class
  - Protected – hides the class's members from classes that exist outside of the class's package
    - accessed within the declaring class, any subclass, or class in the same package as the declaring class
  - Package (Default) – members can be accessed within the declaring class and any class in the same package
    - to have package scope, a member is declared *without* any of the keywords private, protected and public

Visibility Modifiers: Example



Java Visibility Modifiers and Access Privileges

If a member is declared as...	then is the member accessible within...			
	its own class?	any class in its own package?	any subclass in a different package?	any non-subclass in a different package?
public	YES	YES	YES	YES
protected	YES	YES	YES*	NO
package	YES	YES	NO	NO
private	YES	NO	NO	NO

*When a class, S, extends another class, B, every instance of S contains all the members declared in B, regardless of the visibility modifiers (e.g., private, protected, package, or public) on those members. Even though all instances of S contain all the members declared in B, instances of S cannot always refer to all of B's members

Members Visibility Modifiers

- A class member that is declared as `public` can be accessed from any class that can access the class of the member
  - We expose methods that are part of the interface of the class by declaring them as public
- A class member that is declared as `private`, can be accessed only by code that is within the class of this member
  - We hide the internal implementation of the class by declaring its data variables and auxiliary methods as private
  - Data hiding is essential for encapsulation
- Members that are declared without a visibility modifier are said to have `default` visibility
- Note: If a class has a `package` scope then even if its members are `public` there visibility is only at the same `package`

Visibility Modifiers: Example

```
// start of source file A.java
Package samplePkg;
class A {                          // Note: Package scope!!!
  private int num;                 // private
  int get() {return num;}          // package
}
class B {                          // Note: Package scope!!!
  A a1 = new A();                  // package
  void m() {                       // package
       int t = a1.get();           // OK get is visible
       int s = a1.num;             // ERROR: num is private
                                   // in A
      }
}
```

Visibility Modifiers: Example

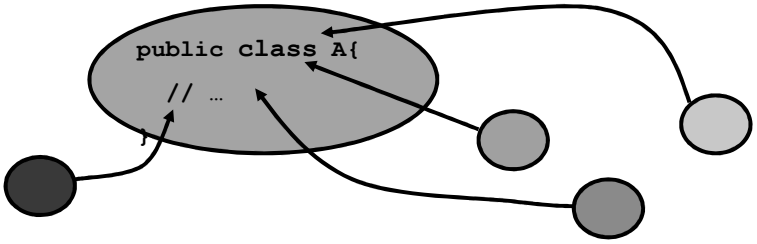
```
// start of source file A.java
Package samplePkg;
class A {                          // Note: Package scope!!!
  int num;                         // package
  public int get() {return num;}// public
}
class B {                          // Note: Package scope!!!
  A a1 = new A();                  // package
  void m() {                       // package
       int t = a1.get();           // OK get is visible
       int s = a1.num;             // OK num is visible
      }
}
```


Public vs. Private Determines what a Class Exports

- We use visibility modifiers to determine which members of a class are part of the external interface (i.e., contract) other objects can access, and which are for internal use only
  - public methods are services that other objects can call on
  - private methods can only be accessed within the class itself, not from outside

Guidelines for Visibility Modifiers

- Classes that define a new type of objects, that are supposed to be used anywhere, should be declared `public`
  - Any other class can use these public classes
- For members:
  - Only make public methods that are in the class's "contract"
  - Make all fields private
  - Make all other "private" methods protected
  - Don't leave off the modifier unless you know about packages



Visibility Modifiers: Example

```
// start of source file A.java
Package samplePkg1;
public class A {                  // public scope
  int num;                        // package
  public int get() {return num;}// public
}
// start of source file B.java
Package samplePkg2;
class B {                         // Note: Package scope!!!
  static A a1 = new A();          // package
  void m() {                      // package
       int t = a1.get();          // OK get is visible
       int s = a1.num;            // ERROR: num is visible
                                  // only within A package
     }
}
```

Errors:
protected class
private class

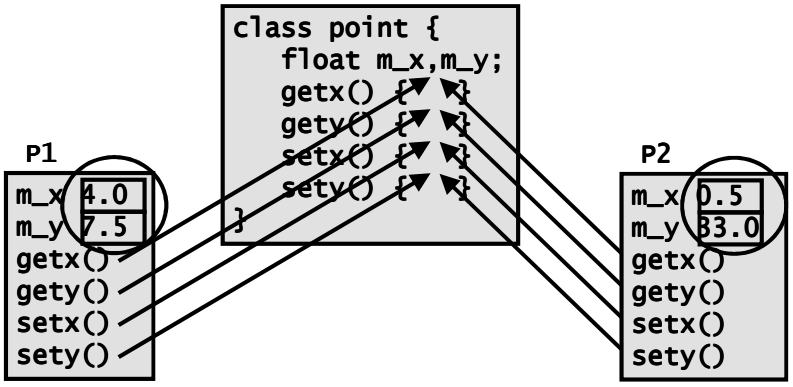
Rules for Protected Class Members

- We declare variables as protected if they will be used often by someone deriving our class, but we still want to hide them from the outside user of our class
  - Avoid defining too many protected variables, we usually want our objects to be as encapsulated as possible
  - Protected methods are usually useful for customizing the behavior of our class by subclassing (subcontracting)
- A protected member is included in the contract of our class
  - You should provide documentation comment to it

```
public class Point {
    /**
     * The coordinates of the point.
     */
    protected int x,y;
    // more functions
}
```


Data Members vs. Method Members

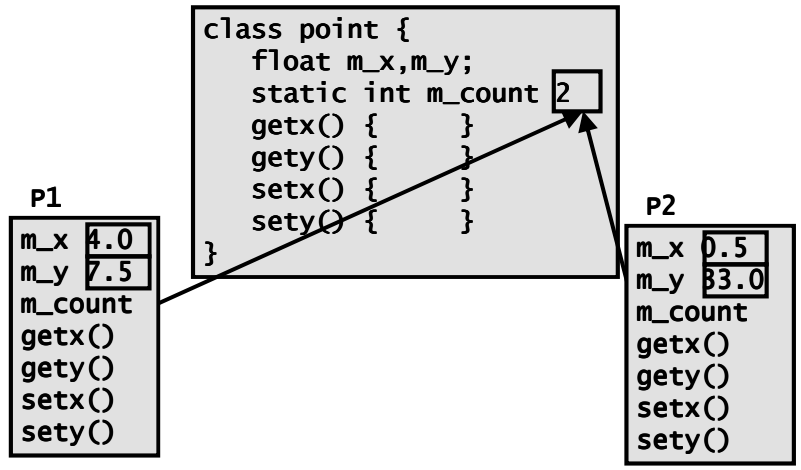
- We use the generic term member to describe an instance variable, an instance method or a constructor of the class instances
- Class instances (objects) share the class method members (the code for execution) but have unique copy of the class data members (variables)



Static Variables and Methods

- Most of the data and methods reside and work on specific objects of a class
  - There are situations in which data and/or methods reside in the class itself
- Only one instance of a static variable exists for the whole class
  - The value of the variable is unique for all instances
  - The variable exist even before any objects of the class are instantiated
  - By default static variables are initialized to 0 or null
- Static methods can operate only on static variables or call other static methods
  - They are not connected to a specific instance and therefore cannot use the '`this`' reference
  - Instance methods can access static variables!

Static Variable Chart



The `Static` Modifier

- All static data/methods are marked with a `static` modifier

```
static <type> variable_name;
static <type> function_name(...);

static  int count;
static  void foo();
```

- The access to a static method is not done through an object but through the class name itself:

```
float i = Math.random();
String s = String.valueOf(i);
```

- If a static method calls another static method of the same class then the class-name can be omitted

Example of Static Variables

- Frequent usage of a static variable is as a counter holding the number of objects of the class

```
class Visitor {
  static int m_count;
  public Visitor() {
      m_count++; }
  }
```

- Example: changes in `m_count` variable as we instantiate Visitor objects:

```
class Theater {
  public static void main(...) {
    //…
    Visitor avi = new Visitor();
    //…
    Visitor beni = new Visitor();
    //…
}
```

→ `m_count = 0`

→ `m_count = 1`

→ `m_count = 2`

Instance Methods vs. `Main`

- The **`main()`** method is static; it is invoked by the system without creating an instance object!

```
public static void main(String[] args)
{
   // ...
}
```

- Definition classes:
  - These classes define new objects
- Container classes:
  - A collection of static methods that are not bound to any particular object
  - These static methods usually have something in common

A Collection of Static Methods

- If no static variables exist static methods can still be used to implement general functionality
  - They just encapsulate a given task, a given algorithm
- We can write a class that is a collection of static methods
  - Such a class isn't meant to define new type of objects
  - It is just used as a library for utilities that are related in some way

From the Java Library: `java.lang.Math`

- The `java.lang.Math` class provides common mathematical functions such as **`sqrt()`**

The Math class can not be subclassed or instantiated

```
// A final class cannot be subclassed
public final class Math {
// A private constructor cannot be invoked
   private Math() {}
   ...
   public static native double sqrt (double a)
                throws ArithmeticException;
}
```

All Math class methods are static class methods. They are invoked as follows: Math.sqrt(55.3)

Math Class Variables

```
/**
 * A library of mathematical methods
 */
public class Math {
    // E static variable
    public static final double E = 2.7182818284590452354;
    // OI static variable
    public static final double PI = 3.14159265358979323846;
    // Computes the trigonometric sine of an angle
    public static native double sin(double x) {
        // ... }
    // Computes the logarithm of a given number
    public static native double log(double x) {
        // ... }
    // ...
}
double x = Math.sin(alpha);
int c = Math.max(a,b);
double y = Math.random();
```

Math Class Methods

Method	Description	Examples

`abs(int x)`
`abs(long x)`
`abs(float x)`
	absolute value of x	if x >= 0 abs(x) is x
		if x < 0 then abs(x) is -x

`ceil(double x)`	rounds x to the smallest integer not less than x	ceil(8.3) is 9 ceil(-8.3) is -8
`floor(double x)`	rounds x to the largest integer not greater than x	floor(8.9) is 8 floor(-8.9) is -9
`log(double x)`	natural logarithm of x	log(2.718282) is 1
`pow(double x, double y)`	x raised to the y power (x^y)	pow(3,4) is 81 pow(16.0, 0.5) is 4.0
`double random()`	generates a pseudorandom number in the interval [0,1)	random() is 0.5551 random()is 0.8712
`round(double x)`	rounds x to an integer	round(26.51) is 27 round (26.499) is 26
`sqrt(double x)`	square root of x	sqrt(4.0) is 2

77

The `final` Modifier

- The final modifier can be used for classes, methods and variables, in each case it has a different meaning
  - A **`final`** variable can be initialized only once (constants)
  - A **`final`** class can not have derived classes
  - A **`final`** method cannot be overridden

78

Method and Class Naming Conventions

- Good Programming Practice
  - Use verbs to name void methods
    - they perform an *action*
  - Use nouns to name methods that return a value
    - they create (return) a piece of data, a *thing*
  - Start class names with a capital letter
  - Start method names with a lower case letter

Method Decomposition

- A method should be relatively small, so that it can understood as a single entity
  - No longer than a page (about 40 lines)
  - A better goal: fits completely on the screen at once (about 20 lines)
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method (i.e. public) of an object may call one or more support methods (i.e., private) to accomplish its goal
- Support methods could call other support methods if appropriate

Multiple `Main` Methods

```
class A
{
      public static void main(...
}
```

```
class B
{
      public static void main(...
}
```

```
class C
{
      public static void main(...
}
```

Each class can have it's own main method

You indicate the one to run when invoking the JVM

This fact becomes critical when we learn to write debug test mains

Methods: Common Mistakes

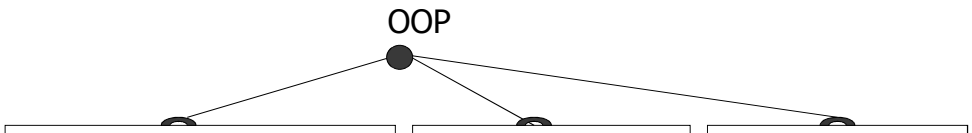
```
public float average (float num1, float num2, float num3);
   {
       float returnVal;
       returnVal =
                  (num1 + num2 + num3)/ 3;
       return (returnVal);
   } // of average
```

- Note the mistakenly placed ending semicolon
  - compiler wrongly views this now as an 'abstract' method (more on abstract methods later)
  - results in unhelpful error message about abstract methods (more on this later)
- You are likely to make this mistake

The Class Method Declaration

accessLevel	Access level for this method.
`static`	This is a class method.
`abstract`	This method is not implemented.
`final`	Method cannot be overridden.
`native`	Method implemented in another language.
`synchronized`	Method requires a monitor to run.
returnType methodName	The return type and method name.
`( paramList )`	The list of arguments.
`throws exceptions`	The exceptions thrown by this method.

Object-Oriented Programming Principles



Encapsulation (class)
-- Information Hiding
-- Interface and Implementations
-- Standardization
-- Access Control mechanisms (private /public etc.)

Inheritance
-- Hierarchy
-- Reusability
-- Extensibility
-- Expressive power
-- Reflects many real-world problems

Polymorphism
-- Many forms of same function
-- Abstract Methods
-- Abstract Classes
-- Interfaces