Lecture 5.1 - 5.2

Writing Classes

Classes

Two viewpoints on classes:

- Client view: how to use an existing class
- Server view: how to design, implement, and maintain classes

Two main uses of classes:

- Auxiliary classes: `Math`, …
- Definition classes: `String`, `Turtle`, `BankAccount`, …

Where Java classes come from:

- The Java standard class library
- Classes that other people write and make available
- Classes that I write.

Outline

- Modularity
- Abstraction
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Method anatomy
  - Class vs. instance methods
  - Types and return values
  - Methods parameters
  - Accessors and mutators
- Related topics
  - Method overloading
  - Revisiting BankAccount
  - Parameter passing
  - Variable life cycle
  - Encapsulation

Modularity

- When building a complex artifact, it always helps to divide the challenge into small, manageable modules
- Each module must have a ...
  - Clearly defined function
  - Contract as to how it interacts with other modules
  - Self-contained design that enables unit-testing and local maintenance
- In particular, when we write a *class*, we have to think how to divide the things that the class is supposed to do into *methods*
- Example: let's look at two versions of the `squareRoot` class from Exercise 3
  - Un-modular version
  - Modular version.

Un-modular version

```
import java.util.Scanner;

public class SquareRoot {
  private static final double EPSILON = 0.1;

  public static void main(String args[]) {
    Scanner scan = new Scanner(System.in);

    // Get the user's input
    System.out.print("Enter a number: ");
    int x = scan.nextInt();

    // Compute the square root
    float root = x / 2;
    while (Math.abs((root * root) - x) > EPSILON) {
      // improve the guess
      root = (root + (x / root)) / 2;
    }

    // Print the result
    System.out.println("The square root is: " + sqrt(x));
  }
}
```

Problems with unmodular code

- The I/O processing and the sqrt computation are "glued" together
- It's hard to tell which part of the program is responsible for run-time bugs
- If we'll want to change the I/O only, we have to change the entire class
- The sqrt services are inaccessible to other clients
- The design is not elegant
- Solution: divide and conquer.

Modular version

```
import java.util.Scanner;

public class SquareRootDemo {

  public static void main(String args[]) {
    Scanner scan = new Scanner(System.in);

    // Get a number from the user and square it
    System.out.print("Enter a number: ");
    int x = scan.nextInt();
    System.out.println("The square root is: " + sqrt(x));
  }

  // Computes the square root function
  public static double sqrt(double x) {
    double precision = 0.1;
    double root = x / 2;
    while (Math.abs((root * root) - x) > precision) {
      root = (root + (x / root)) / 2;
    }
    return root;
  }
}
```

I/O

processing

What have we gained?

- Readability
- Elegance
- Unit testing
- Code re-use
- Parallel development

- Using methods is just one example of modular design
- Modularity is a key design objective that comes up in numerous areas in CS
- Modular design begins at the abstraction level.

Outline

- Modularity
- Abstraction
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Method anatomy
  - Class vs. instance methods
  - Types and return values
  - Methods parameters
  - Accessors and mutators
- Related topics
  - Method overloading
  - Revisiting BankAccount
  - Parameter passing
  - Variable life cycle
  - Encapsulation

Abstraction

- OO program design begins with abstractions
- Abstractions are used to describe objects like car, bank account, turtle, friend, etc.
- To describe an object abstraction, we think about it terms of:
  - What are its attributes? (a data-oriented view)
  - What are its behaviors? (a functional view)

Example: the Clock class

Clock abstraction

A Clock is an object that keeps and reports time using the 24-hour system

Attributes:

- Hours
- Minutes
- Seconds

Behaviors:

- Create a new clock
- Advance the clock 1 second
- Query the clock time
- Etc.

Clock implementation (structure only)

```
public class Clock {

  // The clock state (data)
  private int hours, minutes, seconds;

  // Clock constructor
  public Clock(int h, int m, int s);

  // Advance the clock one second forward
  public void secondElapsed();

  // Query methods
  public int getHours();
  public int getMinutes();
  public int getSeconds();
}
```


Using the Clock (example)

server

```
public class Clock {

  // The clock state
  private int hours, minutes, seconds;

  // Clock constructor
  public Clock(int h, int m, int s);

  // Advance the clock one second forward
  public void secondElapsed();

  // Query methods
  public int getHours();
  public int getMinutes();
  public int getSeconds();
}
```

client

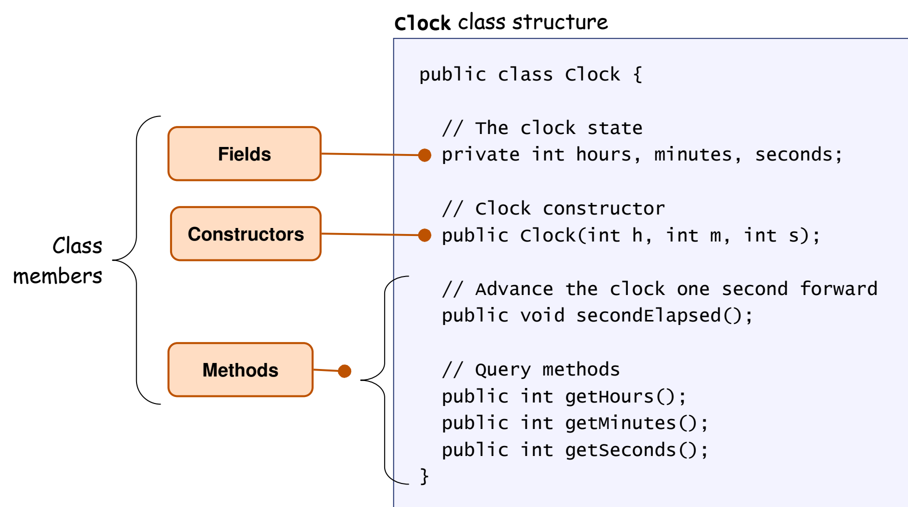
```
public class Someclass {
  ...
  // Create and initialize a clock object
  Clock nyTime = new Clock(23,59,59);

  // Advance the clock 50 seconds
  for (int j = 0; j < 50; j++) {
      nyTime.secondElapsed();
  }
  // Print the clock's current time
  System.out.print("The time in New York is: ");
  System.out.println(nyTime.getHours() + ":" +
    nyTime.getMinutes() + ":" + nyTime.getSeconds());
  ...
}
```

Outline

- Modularity
- Abstraction
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Method anatomy
  - Class vs. instance methods
  - Types and return values
  - Methods parameters
  - Accessors and mutators
- Related topics
  - Method overloading
  - Revisiting BankAccount
  - Parameter passing
  - Variable life cycle
  - Encapsulation

Class members




```
public class Clock {

  // The clock state
  private int hours, minutes, seconds;

  // Clock constructor
  public Clock(int h, int m, int s);

  // Advance the clock one second forward
  public void secondElapsed();

  // Query methods
  public int getHours();
  public int getMinutes();
  public int getSeconds();
}
```

Visibility modifiers:

- `Private` members: visible only to the class code; typically used for fields
- `Public` members: visible to code of other classes; typically used for methods.

Fields

- The object's state (data) is represented by private variables, AKA *instance variables* or *fields*
- Fields are variables: they can be of either primitive or class types
- Each object has a private and separate set of fields
- The fields are typically initialized by constructors.

Clock class structure

```
public class Clock {

  // The clock state
  private int hours, minutes, seconds;

  // Clock constructor
  public Clock(int h, int m, int s);

  // Advance the clock one second forward
  public void secondElapsed();

  // Query methods
  public int getHours();
  public int getMinutes();
  public int getSeconds();
}
```


Constructor

server

```
public class Clock {

  // The clock state
  private int hours, minutes, seconds;

  // Creates a Clock object and initializes
  // its fields using supplied values
  public Clock(int h, int m, int s){
     hours = h;
     minutes = m;
     seconds = s;
  }
  ...
  // Other Clock methods follow
  ...
}
```

- When creating a new object, we normally wish to specify its initial state
- This is done using a *constructor*: a special method which is invoked by the command `Clock c = new Clock(…);`

client

```
Public class SomeClass {
  ...
  Clock c1;
  ...
  c1 = new Clock(12,0,0);
  ...
  Clock c2 = new Clock(10,60,45);
  ...
}
```

This client code causes the server code to create two `Clock` objects.

The `this` keyword

```
public class Clock {

   // The clock state
   private int hours, minutes, seconds;

   // Constructor: one style
   public Clock(int h, int m, int s) {
      hours = h;
      minutes = m;
      seconds = s;
   }

   // Constructor: another style
   public Clock(int hours, int minutes, int seconds) {
      this.hours = hours;
      this.minutes = minutes;
      this.seconds = seconds;
   }
   ...
   // Other Clock methods follow
   ...
}
```

- Inside an instance method, the `this` keyword refers to the object on which the method is currently operating
- `this` can be thought of as a generic object variable, pointing to the current object.

Constructor definition, properly documented using Javadoc

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

  /**
  * Creates a clock and set it to the specified time.
  * If one of the parameters is not in the allowed range,
  * the constructor has no effect on the clock's state.
  * @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 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;
    }
  }
  // ...
}
```

Resulting Clock API, following

`javadoc Clock.java`

Constructor Detail

Clock

```
public Clock(int hours,
             int minutes,
             int seconds)
```

Creates a clock and set it to the specified time. If one of the parameters is not in the allowed range, the constructor has no effect on the clock's state.

Parameters:

`hours` - The hours to be set (0-23)

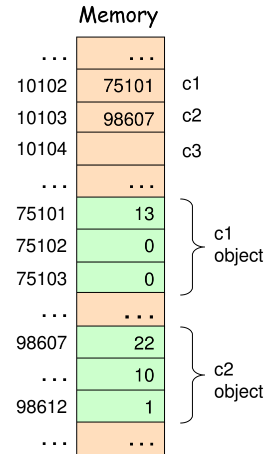
`minutes` - The minutes to be set (0-59)

`seconds` - The seconds to be set (0-59)

Constructor anatomy, behind the scene

Creating a `Clock` object (example)

```
public class SomeClass {
  ...
  Clock c1, c2, c3;
  ...
  c1 = new Clock(13,0,0);
  c2 = new Clock(22,10,1);
  ...
}
```



- When an object variable is declared, only a reference variable is allocated
- The object data is created if and when the constructor is invoked via `new`

Outline

- Modularity
- Abstraction
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Method anatomy
  - Class vs. instance methods
  - Types and return values
  - Methods parameters
  - Accessors and mutators
- Related topics
  - Method overloading
  - Revisiting BankAccount
  - Parameter passing
  - Variable life cycle
  - Encapsulation

Methods

- A *method*, aka *subroutine*, is a stand-alone piece of code designed to carry out a well defined computation or operation
  - Class (static) methods: associated with the class level.
    Example: `sqrt`
  - Instance methods: methods that are associated with objects.
    Example: the `turnRight` method associated with particular `Turtle` objects
- Class methods provide general-purpose functionality
- Instance methods implement object behaviors.

Instance method example

server

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

  // Constructor (omitted)
  ...

  // Advance the clock by 1 second
  public void secondElapsed() {
    if (seconds < 59)
      seconds++;
    else {
      seconds=0;
      if (minutes < 59)
        minutes++ ;
      else {
        minutes = 0;
        hours = hours < 24 ? Hours + 1 : 1;
      }
    }
  }
  // (other methods)
}
```

The desired behaviors of the object are implemented by methods.

Each method is designed to perform one abstract operation

client

```
Public class someClass {
  ...
  Clock c = new Clock(12,0,0);
  ...
  c.secondElapsed();
  ...
}
```

Semantics:
invoke the `secondElapsed` method on `c` (the object that `c` refers to).

Method types and return values

server

```
public class Clock {
  private int hours, minutes, seconds;
  ...
  public void hourElapsed() {
    hours = (hours + 1) % 24;
  }
  ...
  public int getHours() {
    return hours;
  }
  ...
  public String toString() {
    return (hours + "\t" + minutes + "\t" + seconds);
  }
  ...
}
```

client

```
public class SomeClass {
  ...
  Clock c = new Clock(0,0,0);

  // Invoking a void method
  c.hourElapsed(2);

  // Invoking typed methods
  int h = c.getHours();
  System.out.println(c.toString);
  ...
}
```

Void method: has no type and no return value

Typed method: has either a primitive type or an object type.
Returns a value that must conform to the method's type.

Method parameters

- Methods may or may not have parameters
- A parameter is like a local variable which is initialized by the method's caller.

Formal parameters

callee

```
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 invalid
}
```

caller

```
Public class someClass {
  ...
  Clock c1 = new Clock(12, 0, 0);
  ...
  c1.setTime(2, 15, 0);
}
```

Actual parameters (arguments)

By the way …

```
public class Clock {

  // Current time represented by the clock
  private int hours, minutes, seconds;

  // Constructor
  public Clock(int hours, int minutes, int seconds) {
    setTime(hours, minutes, seconds);
  }

  ...

  // Sets the time to given values
  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 invalid
   }
   ...
}
```

Now that we have the `setTime` method, it makes sense to re-define the constructor.

Accessor methods

server

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

  ...

  public int getHours() {
    return hours;
  }

  public int getMinutes() {
    return minutes;
  }

  public int getSeconds() {
    return seconds;
  }

  ...

}
```

Accessor methods: used to query and return information about private variables

Enable controlled access to private variables from the outside.

client

```
public class someClass {
  ...
  Clock c = new Clock(12,0,0);
  ...
  int h = c.getHours();
  ...
  System.out.print(c.getSeconds)
  ...
}
```

Mutator methods

server

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

  ...

  public void setHours(int hours) {
    if ((h > 0) && (h <= 24))
       this.hours = hours;
  }

  public void setMinutes(int minutes) {
    if ((m >= 0) && (m < 60))
       this.minutes = minutes;
  }

  public void setSeconds(int seconds) {
    if ((s >= 0) && (s < 60))
       this.seconds = seconds;
  }

  ...

}
```

Mutator methods: used to set the values of private variables

Enable controlled update of private variables from the outside.

client

```
public class someClass {
  ...
  int deadlineHour = 20;
  ...
  Clock c = new Clock(12,0,0);
  ...
  c.setHours(17);
  ...
  c.setHours(deadlineHour-1);
  ...
  c.setMinutes(scan.nextInt());
  ...
}
```


Method invocation examples

```
public class ClockDemo {

  public static void main(String[] args) {
    Clock c1 = new Clock(12,59,59);
    Clock c2 = new Clock(12,59,59);

    int h = getHours();

    int h = c1.getHours()); // 12

    c1.secondElapsed();

    System.out.println(c1.getHours()); // 1
    System.out.println(c2.getHours()); // 12

    double x = Math.sqrt(c2.getMinutes());
  }
}
```

Error! Which clock are we talking about? (int h = getHours();)

Invoking a void method (c1.secondElapsed();)

Static method invocation (Math.sqrt)

Instance method invocation (c2.getMinutes())

Outline

- Modularity
- Abstraction
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Method anatomy
  - Class vs. instance methods
  - Types and return values
  - Methods parameters
  - Accessors and mutators
- Related topics
  - Method overloading
  - Revisiting BankAccount
  - Parameter passing
  - Variable life cycle
  - Encapsulation

Method overloading

callee

```
public class MyMath {
  ...
  static int sum (int x, int y)) {
      return x + y;
  }
  static int sum (int x, int y, int z)) {
      return x + y + z;
  }
  static double sum (double x, double y)) {
      return x + y;
  }
  ...
}
```

caller

```
public class SomeClass {
  ...
  System.Out.println(MyMath.sum(5,3));
  // will prints 8
  System.Out.println(MyMath.sum(7,2,8));
  // will prints 17
  System.Out.println(MyMath.sum(5.3,4));
  // will print 9.3
  ...
}
```

- Java allows defining different methods with the same name
- Method signature: method name, parameter types, parameter names
- Java knows which method to invoke according to the method signature implied by the caller
- Advantages: Promotes shorter, fewer, and readable method names.

Method overloading

```
public class SquareRootDemo {

public static void main(String args[]) {
  System.out.println(sqrt(17));
  System.out.println(sqrt(17, 0.0001));
}

static double sqrt(double x) {
  return sqrt(x,0.1);
}

static double sqrt(double x, double precision) {
  double root = x / 2;
  while (Math.abs((root * root) - x) > precision) {
    root = (root + (x / root)) / 2;
  }
  return root;
}
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\demo>javac SquareRootDemo.java

D:\demo>java SquareRootDemo
4.124828858612169
4.123105985575862
```


Constructor overloading

server

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

  // Constructors

  public Clock(int hours, int minutes, int seconds) {
    setTime(hours, minutes, seconds);
  }

  public Clock(int hours) {
    this.hours = hours;
    this.minutes = 0;
    this.seconds = 0;
  }

//  Alternative definition:
//  public Clock(int hours) {
//    this(hours, 0 ,0);
//  }

  public Clock() {
    hours = 0;
    minutes = 0;
    seconds = 0;
  }
  ...
}
```

client

```
public class someClass {
  ...
  Clock c1 = new Clock(12,59,59);
  Clock c2 = new Clock(12);
  Clock c3 = new Clock();
  ...
}
```

An overloaded constructor provides multiple ways to set up a new object.

BankAccount class revisited

server

```
public class BankAccount {

  // Account numbers are allocated from 1 onward.
  private static int nextAccountNumber = 1;

  // Fields
  private int number;      // Account number
  private String owner;    // Account owner
  private double balance;  // Current balance

  //  Sets up a new account
  public BankAccount (String owner, double balance) {
    this.number = nextAccountNumber++;
    this.owner = owner;
    this.balance = balance;
  }

// Returns a textual object description
  public String toString () {
    return (number + "\t" + owner + "\t" + (int) balance);
  }
```

(Only a subset of the class members is shown)

client

```
public class BankAccountDemo {

  public static void main(String[] args) {
    BankAccount aliceAccount = new BankAccount("Alice", 0);
    BankAccount bobAccount = new BankAccount("Bob", 100);
    System.out.println(aliceAccount.toString());
    System.out.println(bobAccount.toString());
  }
}
```


BankAccount class revisited

server

```
public class BankAccount {

  // Only a subset of the class members is shown

  private static int bankDeposits = 0;

  // Fields
  private int number;
  private String owner;
  private double balance;

 // Handles a deposit
  public void deposit (double amount) {
    double charge = commission(amount);
    balance = balance + amount - charge;
    bankDeposits += (amount + charge);
  }

  // Handles a withdrawal
  public void withdraw (double amount) {
    double charge = commission(amount);
    balance = balance - amount - charge;
    bankDeposits -= (amount - charge) ;
  }

  // Handles a transfer
  public void transferTo(double amount,
                         BankAccount targetAccount) {
    targetAccount.deposit(amount);
    this.withdraw(amount);
  }
```

client

```
public class BankAccountDemo {

  public static void main(String[] args) {
    BankAccount aliceAccount = new BankAccount("Alice", 0);
    BankAccount bobAccount = new BankAccount("Bob", 100);
    System.out.println(aliceAccount.toString());
    System.out.println(bobAccount.toString());

    aliceAccount.deposit(1000);
    aliceAccount.transferTo(300,bobAccount);
    System.out.println(aliceAccount.toString());
    System.out.println(bobAccount.toString());
  }
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\demo>java BankAccountDemo
1       Alice   0
2       Bob     100
1       Alice   693
2       Bob     398
```

BankAccount class revisited

```
public class BankAccount {

  // Only a subset of the class members is shown
  private static int bankDeposits = 0;
  private static final double COMMISSION_RATE = 0.005;
  private static final int COMMISSION_STEP = 5000;

  private int number;  private String owner;  private double balance;

  // Handles a deposit
  public void deposit (double amount) {
    double charge = commission(amount);
    balance = balance + amount - charge;
    bankDeposits += (amount + charge);
  }

  // Returns the balance
  public double getBalance () {
    return balance;
  }

  // Returns the commission that the bank charges for the amount
  private static double commission (double amount) {
    return ((amount > COMMISSION_STEP) ?
              (amount * COMMISSION_RATE / 2) :
              (amount * COMMISSION_RATE));
  }
```


Call by value / call by reference

callee

Call by value

Call by reference

```
public class BankAccount {

  // Fields
  private int number;
  private String owner;
  private double balance;
  ...
  // Handles a transfer
  public void transferTo(double amount, BankAccount targetAccount) {
    targetAccount.deposit(amount);
    this.withdraw(amount);
    amount = 1000000; // just for the demo
  }
  ...
}
```

caller

```
BankAccount aliceAcc = new BankAccount("Alice", 800);
BankAccount bobAcc = new BankAccount("Bob", 100);
int amount = 500
aliceAcc.transfer(amount, bobAcc);
System.out.println(amount); //  Will print 500.
```

Call by value:

- Used when the parameter is of primitive type
- A value is computed and passed to the method
- The method cannot change the parameter.

Call by reference

- Used when the parameter is of object type
- A pointer to the object is passed to the method
- The method can change the referred object.

Variable life cycle

Static variables

Instance variables

Local variables

- Static variables: created the first time a method from the class is invoked
- Instance variables: created when the object is created and recycled when the object is destroyed
- Local variables: created when the method is invoked and recycled when it returns
- Parameters: same as local variables. Initialized by the arguments supplied by the caller.

```
public class BankAccount {

  private static int lastAccountNumber = 1;
  private static int bankDeposits = 0;

  private int number;
  private String owner;
  private int balance;

  //  Sets up a new account
  public BankAccount (String owner, int balance) {
    this.number = lastAccountNumber++;
    this.owner = owner;
    this.balance = balance;
  }

  // Handles a withdrawal
  public void withdraw (int amount) {
    int balanceTemp = balance - amount
    if (balanceTemp >= 0) {
      balance = balanceTemp;
      bankDeposits -= amount;
    }
    // else ... omitted
    }
  }
```

parameter

Outline

- Modularity
- Abstraction
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Method anatomy
  - Class vs. instance methods
  - Types and return values
  - Methods parameters
  - Accessors and mutators
- Related topics
  - Method overloading
  - Revisiting BankAccount
  - Parameter passing
  - Variable life cycle
  - Encapsulation

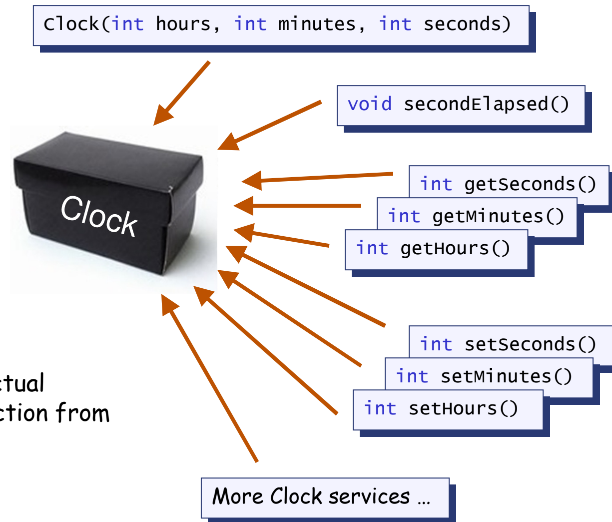
Encapsulation

Clock abstraction:

A Clock is an object that keeps and reports time using the 24-hour system

Required Clock services:

- Create a new clock
- Advance the clock 1 second
- Query the clock time
- Set the clock time

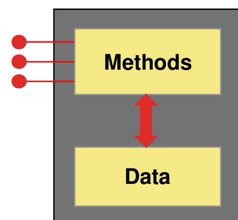


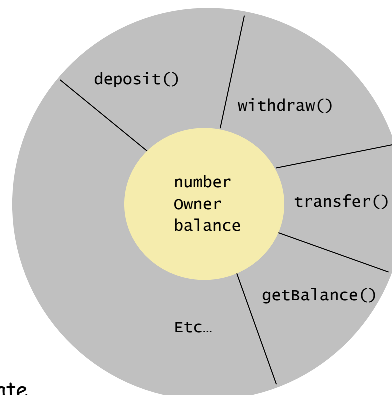
Encapsulation:

- Separating the contractual interface of an abstraction from its implementation.

Encapsulation

Clients





Encapsulation:

- Hiding, or *encapsulating*, the internal state of the object from the outside
- Protects the integrity of the object by preventing users from setting its internal data into an invalid or inconsistent state
- A critically important OO design objective.

Encapsulation and visibility modifiers

	`public`	`private`
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the same class