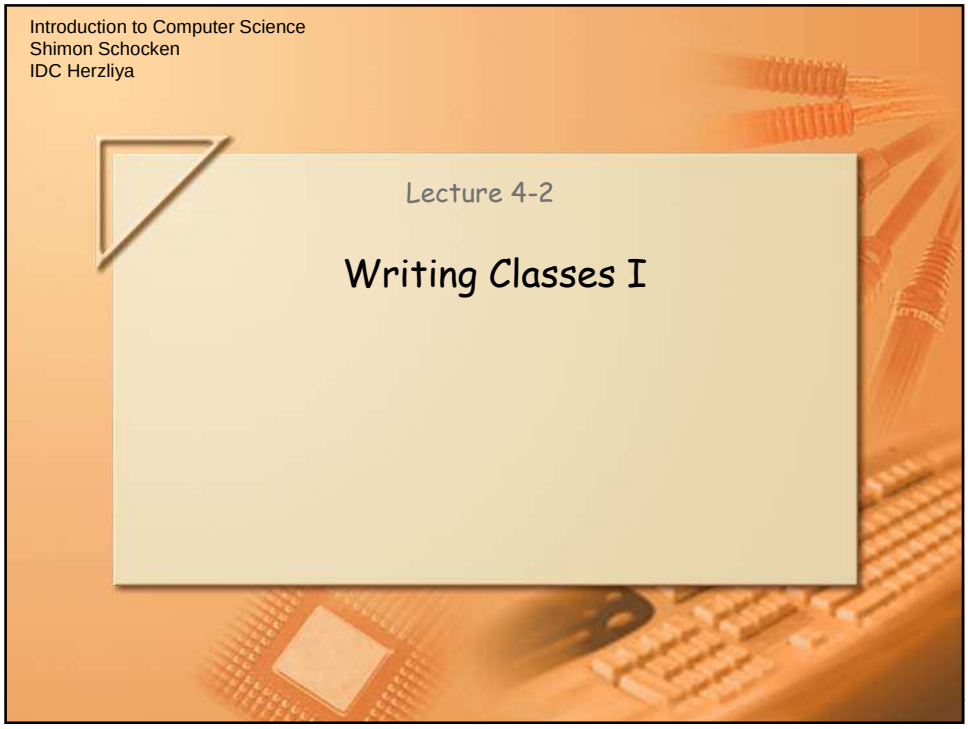


Classes

Two viewpoints on classes:

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

Uses of classes:

- Utility classes: `Math`, `arrays`, …
- ADTs: `String`, `Turtle`, `BankAccount`, …
- More uses of classes (later in the course)

Where Java classes come from:

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

Outline

- Modularity
- Class abstraction
- Class specification
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Accessors and Mutators

Modularity

Modularity is a general systems concept, typically defined as a continuum describing the degree to which a system's components may be separated and recombined. It refers to both the tightness of coupling between components, and the degree to which the "rules" of the system architecture enable (or prohibit) the mixing and matching of components. (Wikipedia)

As software architects, we should always strive to divide our work into small, manageable modules

Each module must have a …

- Clear, simple, and sensible function
- Contract describing its usage
- Self-contained design that enables unit-testing and local maintenance

Example:

- When describing a class abstraction, we think in terms of well-defined operations
- When designing a class, we divide its functionality into well-defined methods
- When writing a method, we divide its code into manageable and well-understood segments

Un-modular code (example)

```
public class SquareRootDemo1 {
    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: ");
        double x = scan.nextDouble();

        // Compute the square root
        double 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: " + root);
    }
}
```

Problems with unmodular code

- The I/O and the processing are mixed together
- It's hard to tell which part of the program is responsible for which bug
- 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: ");
      double x = scan.nextDouble();
      System.out.println("The square root is: " + sqrt(x));
   }

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

I/O

processing

What have we gained?

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

Abstraction

Abstraction is a conceptual process by which higher, more conceptual concepts are derived from the usage and classification of literal (i.e. "real" or "concrete") concepts. Abstractions may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose. For example, abstracting a leather soccer ball to the more general idea of a ball retains only the information on general ball attributes and behavior, eliminating the characteristics of that particular ball. (Wikipedia)

The task of program design begins with abstractions

To describe an object abstraction, we think about it terms of:

- What are the object's attributes? (a data-oriented view, leading to fields)
- What are the object's behaviors? (a functional view, leading to methods)

Abstractions are worked out via requirements analysis

After describing our abstractions, we can move on to write the system's specification

Specification (aka "spec")

A specification is a clear and succinct description of software or hardware that may be used to develop an implementation. It describes *what* the system should do, not (necessarily) *how* the system should do it. Given such a specification, it is possible to demonstrate that a candidate system design is correct with respect to the specification. This has the advantage that incorrect candidate system designs can be revised before a major investment has been made in actually implementing the design. A design (or implementation) cannot be declared "correct" in isolation, but only "correct with respect to a given specification".

Two CS views on specification:

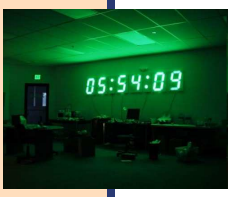
- Formal: a specification is a mathematical artifact that can be analyzed to prove system's correctness (with some caveats)
- Informal: a specification is an informal document that gives the development team enough information on how to design the system.

In this course we take the latter view.

Clocks

`Clock` class specification (partial)

A Clock is an object that keeps and reports time in the format `hours:minutes:seconds`. Hours must be in the range 0 to 23, minutes and seconds must be in the range 0 to 59 each.



Clock's data:

Hours, minutes, seconds

Clock's behaviors:

- Construct a clock and set its time to a given time.
- {Get , set} the clock's {hours , minutes , seconds}. (that's 6 different operations)
- Set the clock's time.
- Advance the clock by one {second, minute, hour}. (that's 3 different operations)
- Determine if this clock's time is later than some other clock's time
- Display the clock's time in the form `hours:minutes:seconds`

Using the Clock (example)

Client code

```
public class ClockTest {
   public static void main (String args[]) {
       Clock c = new Clock(14,0,0);
       System.out.println("The clock time is: " + c);

       // Advances the clock 80 seconds
       for (int j = 0; j < 80; j++)
           c.advanceSecond();

       System.out.println("The clock time is: " + c);
   }
}
```

If the `Clock` class has no `toString` implementation …

```
The time in New York is: Clock@de6ced
The time in New York is: Clock@de6ced
```

Fixed according to the class spec …

```
The clock time is: 14:00:00
The clock time is: 14:01:20
```

Maybe delay this to the next Class Writing lecture (toString)

Outline

- Modularity
- Class abstraction
- Class specification
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Accessors and Mutators

Class anatomy

`Clock` class structure (method signatures)

Fields

Constructors

Methods

```
public class Clock {

   // The clock state
   private int hours, minutes, seconds;
   // More fields ...

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

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

   public int getHours();
   public int getMinutes();
   public int getSeconds();
   // More methods ...

}
```

Class members:

Taken together, the fields and the methods are called the "class members"

Visibility:

Class members can have different visibility outside the class (`private` / `public`).

More about visibility, later.

Fields

- Objects are typically characterized by a set of *attributes*, aka *properties*
- These are implemented using *Fields*, aka *private variables* or *instance variables*
- Taken together, the fields represent the object's state (data)
- Fields are variables: can be of either primitive or class types
- Each object has a private and separate set of field values
- The fields are typically initialized by constructors
- Uninitialized fields are set by the compiler to default values

`Clock` class structure

```
public class Clock {

   // The clock state
   private int hours, minutes, seconds;
   // More fields ...

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

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

   // Accessors
   public int getHours();
   public int getMinutes();
   public int getSeconds();
   // More methods ...
}
```

- Within the class code, the field `x` of object `c` is accessible using the syntax `c.x`; where `this.x` or simply `x` refer to field `x` of the current object.

Constructors

server

```
public class Clock {

   private int hours, minutes, seconds;

   public Clock(int h, int m, int s) {
       setTime(h, m, s);
   }

   public Clock(); {
       setTime(0, 0, 0);
   }
   // ...
}
```

- When creating a new object, we normally wish to specify its initial state
- This is done using constructors
- Constructors can be overloaded
- If you don't write a constructor, the compiler inserts a default constructor into the class code

Best practice: always write your own constructor(s), even if they do nothing.

```
// Default constructor, inserted
// to the class code by the
// compiler if the programmer did
// not define a constructor.

public Clock() {}
```

client

```
Public class SomeClass {
  ...
  Clock c1, c2;
  ...
  c1 = new Clock(16,30,20);
  c2 = new Clock();
  Clock c3 = new Clock(12,0,0);
  ...
}
```

Constructors: several ways to define a Clock constructor

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

```
public Clock(int h, int m, int s) {
    hours = h;
    minutes = m;
    seconds = s;
}
```

(In these examples we assume that the class has no `setTime` method)

```
public Clock(int h, int m, int s) {
    this.hours = h;
    this.minutes = m;
    this.seconds = s;
}
```

this: a generic object variable, pointing to the current object.

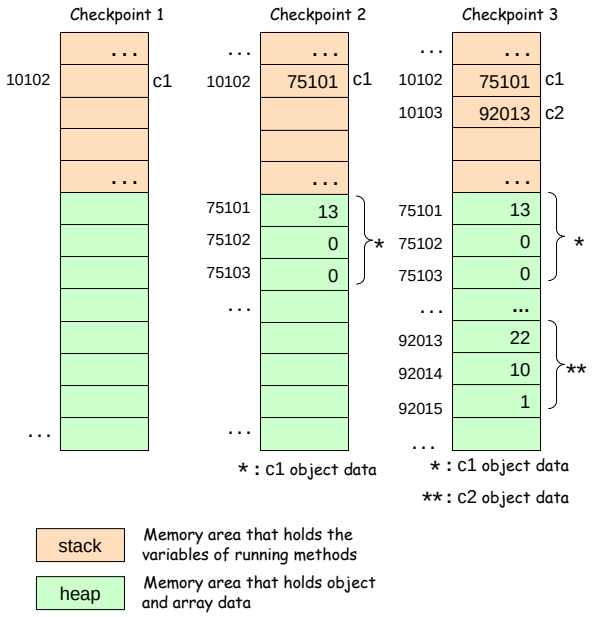
```
public Clock(int hours, int minutes, int seconds) {
    this.hours = hours;
    this.minutes = minutes;
    this.seconds = seconds;
}
```

Most readable

Constructor anatomy: behind the scene

Creating `Clock` objects

```
public class SomeClass {
  ...
  Clock c1;
  // Checkpoint 1
  ...
  c1 = new Clock(13,0,0);
  // Checkpoint 2
  ...
  Clock c2 = new Clock(22,10,1);
  // Checkpoint 3
  ...
}
```



- When an object variable is declared, only the reference variable is allocated
- Memory for the object proper is allocated only if and when a constructor is invoked using `new`.

stack — Memory area that holds the variables of running methods

heap — Memory area that holds object and array data

Outline

- Modularity
- Class abstraction
- Class specification
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Accessors and Mutators

Methods

Method: a stand-alone piece of code, designed to carry out a well defined computation or operation

Also called "subroutine" or "function" in other programming languages

Instance methods:

Implement object behaviors

Operate on the current object

Class (static) methods:

Perform some general-purpose functionality

Not associated with any particular object.

Best practice advice:

A class should contain either instance methods, or static methods, but not both.

Method definition and invocation (examples)

server

```
public class Clock {

    private int hours, minutes, seconds;
    ...

    public int getHours() {
        return hours;
    }

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

    public void advanceSecond() {
        if (seconds < 59) {
            seconds++;
            return;
        }
        else
            seconds = 0;
        if (minutes < 59)
            minutes++;
        else {
            minutes = 0;
            hours = hours < 23 ? hours+1 : 0;
    }
    ...
}
```

The desired behaviors of the object are implemented by methods

Each method is designed to perform one well-defined abstract operation

client

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

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

Method parameters

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

callee

Formal parameters

```
public void setTime (int h, int m, int s) {
    setHours(h);
    setMinutes(m);
    setSeconds(s);
}
```

caller

```
Public class someClass {
  ...

  Clock c1 = new Clock(12, 0, 0);
  ...

  c1.setTime(2, 15, 0);
  ...
}
```

Actual parameters (arguments)

Variable kinds in Java (wrap-up)

- Class (static) variable
- Private (field) variable
- Local variable
- Parameter

Method types and the `return` command

server

```
public class Clock {

  private int hours, minutes, seconds;
  ...

  public int getHours() {
     return hours;
  }

  public void setHours(int h) {
     this.hours = h;
  }

  public String toString() {
     return (hours + ":" + minutes + ":" + seconds);
  }
  ...
}
```

client

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

  // Calling a void method
  c.setHours(17);

  // Calling a typed method
  int h = c.getHours();

  // More examples
  System.out.println(c);
  int bla = Math.sqrt(c.getHours())
```

If a method is of type int, then its return value can be treated as an int variable and can play any role that an int variable is allowed to play in Java expressions.

Same for any other method type.

Void method: has no type and no return value

Typed method: has either a primitive type or an object type.
Must return a value that conforms to the method's type.

Referring to objects and fields within the class code

server

```
public class Clock {
  private int hours, minutes, seconds;
  ...
  public boolean laterThan(Clock c) {
      if (hours > c.hours)
          return true;
      if ((hours == c.hours) && (minutes > c.minutes))
          return true;
      if ((hours == c.hours) && (minutes == c.minutes) && (seconds > c.seconds))
          return true;
      return false;
    }
```

client

```
public class SomeClass {
...
  Clock c1 = new Clock(22,30,20);
  Clock c2 = new Clock(22,30,5);
  System.out.println(c1.laterThan(c2));   // true
  System.out.println(c1.laterThan(new Clock(22,20,20))); // true
  System.out.println(c1.laterThan(new Clock(21,30,20))); // true
  System.out.println(c1.laterThan(new Clock(22,40,20))); // false
...
}
```

Outline

- Modularity
- Class abstraction
- Class specification
- Class anatomy
  - Fields
  - Constructors
  - Methods
- Accessors and Mutators

Accessor methods (getters)

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 the object state, e.g. return values of private variables

Enable controlled access to the object's state from the outside.

client

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

Best practice advice:

Define all fields as private and write accessor methods to facilitate their access.

An exception: record types (later)

Mutator methods (setters)

server

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

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

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

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

Best practice advice: Always use private variables to represent object's attributes, and write setter methods to test and set their values.

Mutator methods: used to set the values of private variables

Enable controlled update of the object's state from the outside.

client

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

Instead of (if the fields were declared public):

```
  c.hours = deadlineHour - 1;
  c.minutes = 1;
  c.seconds = scan.nextInt());
```