3.1 Objects
A Foundation for Programming

Any program you might want to write

- Objects
- Functions and modules
- Graphics, sound, and image I/O
- Arrays
- Conditionals and loops
- Math
- Text I/O
- Primitive data types
- Assignment statements

Create your own data types
Data Types: set of values and associated operations

Primitive Types:
- values map directly to the machine representation
- ops map directly to machine instructions

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Set of Values</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>not, and, or, xor</td>
</tr>
<tr>
<td>int</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
<td>add, subtract, multiply</td>
</tr>
<tr>
<td>double</td>
<td>any of $2^{64}$ possible reals</td>
<td>add, subtract, multiply</td>
</tr>
</tbody>
</table>

We want to write programs that handle other data types
- colors, pictures, strings, input streams, ...
- complex numbers, vectors, matrices, polynomials, ...
- points, polygons, charged particles, celestial bodies, ...
Objects

**Objects:** represent values and operations for more complex data types

– Object variables are called **fields**
– Object operations are called **methods**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Set of Values</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>24 bits</td>
<td>get red component, brighten</td>
</tr>
<tr>
<td>Picture</td>
<td>2D array of colors</td>
<td>get/set color of pixel (i, j)</td>
</tr>
<tr>
<td>String</td>
<td>sequence of characters</td>
<td>length, substring, compare</td>
</tr>
</tbody>
</table>

Objects are said to **encapsulate** (hide) its detail

– How an object is implemented is not important
– What it does is important

**Objects** can be created and referenced with variables
Object-Oriented Programming

Programming paradigm that views a program as a collection of interacting objects
- In contrast, the conventional model views the program as a list of tasks (subroutines or functions)

We’ll talk about how to:
- Create your own data types (set of values and operations)
- Use objects in your programs (e.g., manipulate objects)

Why would I want to use objects in my programs?
- Simplify your code
- Make your code easier to modify
- *Share an object with a friend*
The String Object

**public class String (Java string data type)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String(String s)</td>
<td>create a string with the same value as s</td>
</tr>
<tr>
<td>int length()</td>
<td>string length</td>
</tr>
<tr>
<td>char charAt(int i)</td>
<td>i\textsuperscript{th} character</td>
</tr>
<tr>
<td>String substring(int i, int j)</td>
<td>i\textsuperscript{th} through (j-1)\textsuperscript{st} characters</td>
</tr>
<tr>
<td>boolean contains(String sub)</td>
<td>does string contain sub as a substring?</td>
</tr>
<tr>
<td>boolean startsWith(String pre)</td>
<td>does string start with pre?</td>
</tr>
<tr>
<td>boolean endsWith(String post)</td>
<td>does string end with post?</td>
</tr>
<tr>
<td>int indexOf(String p)</td>
<td>index of first occurrence of p</td>
</tr>
<tr>
<td>int indexOf(String p, int i)</td>
<td>index of first occurrence of p after i</td>
</tr>
<tr>
<td>String concat(String t)</td>
<td>this string with t appended</td>
</tr>
<tr>
<td>int compareTo(String t)</td>
<td>string comparison</td>
</tr>
<tr>
<td>String replaceAll(String a, String b)</td>
<td>result of changing as to bs</td>
</tr>
<tr>
<td>String[] split(String delim)</td>
<td>strings between occurrences of delim</td>
</tr>
<tr>
<td>boolean equals(String t)</td>
<td>is this string’s value the same as t’s?</td>
</tr>
</tbody>
</table>
Constructors and Methods

To construct a new object:
- Use keyword `new` (to invoke constructor)
- Use name of data type (to specify which type of object) with associated parameters for the constructor

To apply an operation:
- Use name of object (to specify which object)
- Use the dot operator (to access a member of the object)
- Use the name of the method (to specify which operation)

```java
String s;
s = new String("Hello, World");
System.out.println(s.substring(0, 5));
```
In defining your own objects with classes:

- Classes are blueprints or prototypes for new objects.
- Classes define all field and method declarations, which are repeated for each new object created.
- Using a class to create a new object is called instantiating an object, creating a new object instance of the class.
- Classes often model real-world items.
Bouncing Ball Object

• What do we want to have the ball do? (i.e., what methods should it have?)

• What initial parameters should we specify in the constructor?
Bouncing Ball Object

• What do we want to have the ball do? (i.e., what methods should it have?)
  – void draw() : “Ball, draw thyself!”
  – void update() : simulate the ball’s motion

• What initial parameters should we specify in the constructor?
Bouncing Ball Object

• What do we want to have the ball do? (i.e., what methods should it have?)
  – void draw() : “Ball, draw thyself!”
  – void update() : simulate the ball’s motion

• What initial parameters should we specify in the constructor
  – Ball (int x, int y) : creates a ball at (x, y)

These methods constitute the ball’s API
Bouncing Ball Object

Given only the API, we can use the object in a program:

```java
static Ball[] balls = new Ball[20];

public static void setup() {
    // Create all new Ball objects
    for (int i = 0; i < balls.length; i++) {
        balls[i] = new Ball(Math.random(), Math.random());
    }
}

don't break

public static void draw() {
    StdDraw.clear(StdDraw.WHITE);
    for (int i = 0; i < balls.length; i++) {
        balls[i].update();  // Methods of objects stored in the array
        balls[i].draw();    // are accessed using dot-notation.
    }
}
```

Declare an array of Balls.

New objects are created with the `new` keyword.

Methods of objects stored in the array are accessed using dot-notation.
Bouncing Ball Object Implementation

• What fields should the ball have?
  (i.e., what does it need to know about itself?)
  – position (x,y)
  – velocity (dx, dy)
  – acceleration due to gravity (ay)
  – size, color, etc...

• The class Ball is implemented in the same file
  (BouncingBallObjectDemo.java)
// Defining a new object
public class MyObjectName {

  // All field variable declarations go here.
  // Field variables should be private.

  /* Define a special function-like statement called
   * the object’s **constructor**.
   * Its name is same as the class name,
   * with no return value.
   */
  public MyObjectName( optional arguments ) {
    // Perform all initialization here
  }

  // Declare all method functions here

}
/ A Ball Class
public class Ball {

   // Fields
   private double ay = 0.002;   // y acceleration (gravity)
   private double x;            // x position
   private double y;            // y position
   private double dx;           // x velocity
   private double dy;           // y velocity
   private double radius = 0.05;

   // Constructor
   public Ball() {
      x = StdRandom.uniform(radius, 1 - radius);
      y = StdRandom.uniform(0.5, 1);
      dx = StdRandom.uniform(-0.03, 0.03);
      dy = StdRandom.uniform(0.0, 0.05);
   }

   ...
   }
}
private boolean canBounceOffWalls = true;
private boolean canBounceOffFloors = true;

// Methods
public void update() {
    // Move ball
    x += dx;
    y -= dy;
    dy += ay;

    // Bounce off walls and floor
    if (canBounceOffWalls && (x < radius || x > (1 - radius))) {
        dx = -dx;
        canBounceOffWalls = false;
    }
    if (canBounceOffFloors && y < radius) {
        dy = -0.9*dy;
        canBounceOffFloors = false;
    }

    // reset ready-to-bounce flags
    if (x >= radius && x <= (1 - radius))   canBounceOffWalls = true;
    if (y >= radius)    canBounceOffFloors = true;
}

public void draw() {
    PennDraw.filledCircle(x, y, radius);
}
Comparing Declarations and Initializers

```java
int i;
int j = 3;
float f = 0.1;
float[] f2 = new float[20];
String s1 = "abc";
String s2 = new String("abc");
Ball b = new Ball();

Ball[] b2 = new Ball[20];
for (int i = 0; i < b2.length; i++) {
    b2[i] = new Ball();
}
```
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0</td>
</tr>
<tr>
<td>C1</td>
<td>0</td>
</tr>
<tr>
<td>C2</td>
<td>0</td>
</tr>
<tr>
<td>C3</td>
<td>0</td>
</tr>
<tr>
<td>C4</td>
<td>0</td>
</tr>
<tr>
<td>C5</td>
<td>0</td>
</tr>
<tr>
<td>C6</td>
<td>0</td>
</tr>
<tr>
<td>C7</td>
<td>0</td>
</tr>
<tr>
<td>C8</td>
<td>0</td>
</tr>
<tr>
<td>C9</td>
<td>0</td>
</tr>
<tr>
<td>CA</td>
<td>0</td>
</tr>
<tr>
<td>CB</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
</tr>
</tbody>
</table>

main memory
(64-bit machine)
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0.50</td>
</tr>
<tr>
<td>C1</td>
<td>0.50</td>
</tr>
<tr>
<td>C2</td>
<td>0.05</td>
</tr>
<tr>
<td>C3</td>
<td>0.01</td>
</tr>
<tr>
<td>C4</td>
<td>0.03</td>
</tr>
<tr>
<td>C5</td>
<td>0</td>
</tr>
<tr>
<td>C6</td>
<td>0</td>
</tr>
<tr>
<td>C7</td>
<td>0</td>
</tr>
<tr>
<td>C8</td>
<td>0</td>
</tr>
<tr>
<td>C9</td>
<td>0</td>
</tr>
<tr>
<td>CA</td>
<td>0</td>
</tr>
<tr>
<td>CB</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
</tr>
</tbody>
</table>

registers | main memory (64-bit machine)
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0.60</td>
</tr>
<tr>
<td>C1</td>
<td>0.52</td>
</tr>
<tr>
<td>C2</td>
<td>0.05</td>
</tr>
<tr>
<td>C3</td>
<td>0.01</td>
</tr>
<tr>
<td>C4</td>
<td>0.03</td>
</tr>
<tr>
<td>C5</td>
<td>0</td>
</tr>
<tr>
<td>C6</td>
<td>0</td>
</tr>
<tr>
<td>C7</td>
<td>0</td>
</tr>
<tr>
<td>C8</td>
<td>0</td>
</tr>
<tr>
<td>C9</td>
<td>0</td>
</tr>
<tr>
<td>CA</td>
<td>0</td>
</tr>
<tr>
<td>CB</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
</tr>
</tbody>
</table>

registers

main memory
(64-bit machine)
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();
b2 = b1;
b2.update();
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0.60</td>
</tr>
<tr>
<td>C1</td>
<td>0.52</td>
</tr>
<tr>
<td>C2</td>
<td>0.05</td>
</tr>
<tr>
<td>C3</td>
<td>0.01</td>
</tr>
<tr>
<td>C4</td>
<td>0.03</td>
</tr>
<tr>
<td>C5</td>
<td>0</td>
</tr>
<tr>
<td>C6</td>
<td>0</td>
</tr>
<tr>
<td>C7</td>
<td>0.50</td>
</tr>
<tr>
<td>C8</td>
<td>0.50</td>
</tr>
<tr>
<td>C9</td>
<td>0.07</td>
</tr>
<tr>
<td>CA</td>
<td>0.04</td>
</tr>
<tr>
<td>CB</td>
<td>0.04</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
</tr>
</tbody>
</table>

registers | main memory (64-bit machine)
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```

**C7 – CB** can be reused for other variables. Known as **garbage collection** in Java.
Object References

- Allow client to manipulate an object as a single entity
- Essentially a machine address (pointer)

```java
Ball b1 = new Ball();
b1.update();
b1.update();

Ball b2 = new Ball();
b2.update();

b2 = b1;
b2.update();
```

Moving `b2` also moves `b1` since they are **aliases** that reference the same object.
Arguments to methods are always passed by value.

- Primitive types: passes copy of value of actual parameter.
- Objects: passes copy of reference to actual parameter.

```java
public class PassByValue {
    static void update(int a, int[] b, String c) {
        a = 7;
        b[3] = 7;
        c = "seven";
        System.out.println(a + " " + b[3] + " " + c);
    }

    public static void main(String[] args) {
        int a = 3;
        int[] b = { 0, 1, 2, 3, 4, 5 };
        String c = "three";
        System.out.println(a + " " + b[3] + " " + c);
        update(a, b, c);
        System.out.println(a + " " + b[3] + " " + c);
    }
}
```
Encapsulation
Access Control

• Encapsulation is implemented using **access control**.
  – Separates interface from implementation
  – Provides a boundary for the client programmer

• Visible parts of the class (the **interface**)
  – can be used and/or changed by the client programmer.

• Hidden parts of the class (the **implementation**)
  – Can be changed by the class creator without impacting any of the client programmer’s code
  – Can’t be corrupted by the client programmer
Access Control in Java

- **Visibility modifiers** provide access control to instance variables and methods.
  - *public* visibility - accessible by everyone, in particular the client programmer
    - A class’ interface is defined by its public methods.
  - *private* visibility - accessible only by the methods within the class
    - Two others—*protected* and *package*—later
Good Programming Practice

• Combine methods and data in a single class
• Label all instance variables as `private` for information hiding
  – The class has complete control over how/when/if the instance variables are changed
  – Fields primarily support class behavior
• Minimize the class’ public interface
Using **this**

You can think of **this** as an implicit private reference to the current instance.

Date
=== public ====
Date()
int getYear()
...
=== private ====
int month
int day
int year
Date this ...

Date b1 = new Date();

```
b1

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>1</td>
</tr>
<tr>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td>C2</td>
<td>1900</td>
</tr>
<tr>
<td>C3</td>
<td>C0</td>
</tr>
<tr>
<td>C4</td>
<td>?</td>
</tr>
<tr>
<td>C5</td>
<td>?</td>
</tr>
<tr>
<td>C6</td>
<td>?</td>
</tr>
</tbody>
</table>
```

Main memory (64-bit machine)

Registers

Note that `b1.year` and `b1.this.year` refer to the same field.
Overloaded Constructors

```java
public class Date {
    private int month;  // 1 - 12
    private int day;    // 1 - 31
    private int year;   // 4 digits

    // no-argument constructor
    public Date() {
        month = 1;
        day = 1;
        year = 1900;
    }

    // alternative constructor
    public Date(int month, int day, int year) {
        this.month = month;
        this.day = day;
        this.year = year;
    }

    // 1 Jan 1900
    Date d1 = new Date();

    // 30 Oct 2013
    Date d2 = new Date(10, 30, 2013);
}
```
Accessors & Mutator

• Class *behavior* may allow access to, or modification of, individual private instance variables.

• Accessor method
  – retrieves the value of a private instance variable
  – conventional to start the method name with *get*

• Mutator method
  – changes the value of a private instance variable
  – conventional to start the name of the method with *set*

• Gives the client program *indirect* access to the instance variables.
Question: Doesn’t the use of accessors and mutators defeat the purpose of making the instance variables *private*?

Answer: **No**

- The class implementer decides which instance variables will have accessors.
- Mutators can:
  - validate the new value of the instance variable, and
  - decide whether or not to actually make the requested change.
public class Date {
    private int month;   // 1 - 12
    private int day;  // 1 - 31
    private int year;  // 4-digit year

    // accessors return the value of private data
    public int getMonth()  { return month; }

    // mutators can validate the new value
    public boolean setMonth(int month) {
        if (1 <= month && month <= 12) {
            this.month = month;
            return true;
        } else // this is an invalid month
            return false;
    }

    // rest of class definition follows
}
Accessor/Mutator Caution

• In general you should NOT provide accessors and mutators for all private instance variables.
  
  – Recall that the principle of encapsulation is best served with a *limited class interface*. 
Private Methods

• Methods may be private.
  – Cannot be invoked by a client program
  – Can only be called by other methods within the same class definition
  – Most commonly used as “helper” methods to support top-down implementation of a public method
public class Date {
    private int month;   // 1 - 12
    private int day; // 1 - 31
    private int year; // 4-digit year

    // accessors return the value of private data
    public int getMonth()  { return month; }

    // mutators can validate the new value
    public boolean setMonth(int month) {
        if (isValidMonth(month)) {
            this.month = month;
            return true;
        } else { // this is an invalid month
            return false;
        }
    }

    // helper method - internal use only
    private boolean isValidMonth(int month) {
        return 1 <= month && month <= 12;
    }
}
Static and Final
Static Variable

• A static variable belongs to the class as a whole, not just to one object.

• There is only one copy of a static variable per class.
  – All objects of the class can read and change this static variable.

• A static variable is declared with the addition of the modifier static.
  
  ```java
  static int myStaticVariable = 0;
  ```
Static Constants

- A **static constant** is used to symbolically represent a constant value.

  - The declaration for a static constant includes the modifier `final`, which indicates that its value cannot be changed:

    ```java
    public static final float PI = 3.142;
    ```

- It is not necessary to instantiate an object to access a static variable, constant or method.

- When referring to such a constant outside its class, use the name of its class in place of a calling object.

  ```java
  float radius = MyClass.PI * radius * radius;
  ```
Rules for Static Methods

• Static methods have no calling/host object (they have no `this`).

• Therefore, static methods **cannot**:
  – Refer to any instance variables of the class
  – Invoke any method that has an implicit or explicit `this` for a calling object

• Static methods **may** invoke other static methods or refer to static variables and constants.

• A class definition may contain both static methods and non-static methods.
main is a Static Method

Note that the method header for main( ) is

```java
public static void main(String[] args)
```

Being static has two effects:

- main can be executed without an object.
- “Helper” methods called by main must also be static.
Any Class Can Have a main( )

• Every class can have a public static method name main( ).

• Java will execute main in whichever class is specified on the command line.

   java <className>

• A convenient way to write test code for your class.
Static Review

• Given the skeleton class definition below

```java
public class C {
    public int a = 0;
    public static int b = 1;

    public void f() {...}
    public static void g() {...}
}
```

• Can body of f() refer to a?
• Can body of f() refer to b?
• Can body of g() refer to a?
• Can body of g() refer to b?
• Can f() call g()?
• Can g() call f()?

For each, explain why or why not.