Data Abstraction (Chap 6)

- In Java there are three types of data values:
  - primitive data values (int, double, boolean, etc.)
  - arrays (actually a special type of object)
  - objects
- Objects in a program are used to represent "real" (and sometimes not so real) objects from the world around us.

Objects

- An object might represent a string of characters, a planet, a type of food, a student, an employee, a piece of email, ... anything that can't be (easily) represented by a primitive value or an array.
- Just as 3 is a primitive value of type int, every object must also have a type. These types are called classes.

Classes

A class describes a set of objects.
- It specifies what information will be used to represent an object from the set (e.g. name and salary for an employee).
- It also specifies what operations can be performed on such an object (get the name of the student, send an email message).

The class String

- String is a standard Java class. Values from the class String are called objects, so "hello" is an object from the class String.
- We can also say "hello" is an instance of the class String.
- The class String has instance methods that operate on an instance of class String. For example: length() and charAt().

```java
// HelloWorld3.java - variable declarations
// from HelloWorld2 in Chapter 2
class HelloWorld3 {
    public static void main (String[] args) {
        String word1; // variable
        String word2, sentence;
        word1 = "Hello, ";
        word2 = "world!";
        sentence = word1.concat(word2);
        System.out.println(sentence);
        System.out.println(sentence.length());
        System.out.println(sentence.charAt(0));
    }
}
```

Put some instructor notes here for what if... reassign word1.
String is a bit special
Because strings are so common, Java has special two pieces of special syntax for the class String.
– There is syntactic support for string concatenation.
– There is syntactic support for creating string literals.

String concatenation
The operator + is overloaded to implement concatenation of strings.
"hello, " + "world"
is equivalent to
"hello, " .concat("world")

String literals
String literals are supported.
String s = "hello"
is equivalent to
char[] temp={'h','e','l','l','o'};
String s = new String(temp);

Elements of a Simple Class
• A class describes the data values used to represent an object and any operations that can be performed on that object.
• The data values are stored in instance variables, also known as fields, or data members.
• The operations are described by instance methods, sometimes called procedure members.

Making Your Own Classes

class Person {
    int age;
    String name;
    char gender;
}
class PersonTest {
    public static void main(String[] args) {
        Person john = new Person();
        Person jane = new Person();
        john.age = 19;
        john.name = "John Doe";
        john.gender = 'M';
        jane.age = 20;
        jane.name = "Jane Doe";
        jane.gender = 'F';
        printPerson(jane);
        printPerson(john);
    }
}
static void printPerson(Person p) {
    System.out.println("Name: " + p.name);
    System.out.println("Age: " + p.age);
    System.out.println("Gender: " + p.gender);
}

import java.util.Random;
class Dice {
    // constructor initializes the object
    Dice(int seed) {
        roller = new Random(seed);
    }
    // mutator instance method operates on the object
    void roll() {
        die1 = roller.nextInt(6) + 1;
        die2 = roller.nextInt(6) + 1;
    }
    // accessor instance method gets values from the object
    int getTotal() {
        return die1 + die2;
    }
    // inherited method used to turn object into a String
    public String toString() {
        return die1 + ", " + die2;
    }
    private int die1, die2;
    private Random roller;
}

import java.util.Scanner;
class DiceTest {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the seed.");
        Dice dice = new Dice(input.nextInt());
        System.out.println("How many times should I roll?");
        int count = input.nextInt();
        while(count > 0) {
            dice.roll();
            System.out.println("You rolled " + dice);
            System.out.println("The total is " + dice.getTotal());
            count--;
        }
    }
}

Object Storage (The Heap)
Local Variable Storage
(The Stack)

Call by Value vs Call by Ref

- Although you can modify an object that is passed to a method, you still cannot modify a variable that is a reference type variable (a class or array).
- There is a difference between modifying an object, and modifying a variable, when the variable is a reference (anything other than a primitive type in Java).
class ModifyParameters {
    public static void main(String[] args) {
        Dice dice = new Dice(1234);
        dice.roll();
        System.out.println("dice is now " + dice);
        modify(dice);
        System.out.println("dice is now " + dice);
    }
    static void modify(Dice dice) {
        dice = new Dice(456);
        System.out.println("dice in modify is " + dice);
    }
}

class ModifyParameters2 {
    public static void main(String[] args) {
        Dice dice = new Dice(1234);
        dice.roll();
        System.out.println("dice is now " + dice);
        modifyWorks(dice);
        System.out.println("dice is now " + dice);
    }
    static void modifyWorks(Dice dice) {
        dice.roll();
        System.out.println("dice in modify is " + dice);
    }
}

class Counter {
    int value;                      // instance variable
    void reset() { value = 0; }     // mutator method
    int get()    { return value;}   // accessor method
    void click() { value = (value + 1) % 100;}  
}

// CounterTest.java - demonstration of class Counter
class CounterTest {
    public static void main(String[] args) {
        Counter c1 = new Counter();  // create a Counter
        Counter c2 = new Counter();  // create another
        c1.click();   // increment Counter c1
        c2.click();   // increment Counter c2
        c2.click();   // increment Counter c2 again
        System.out.println("Counter1 value is " + c1.get());
        System.out.println("Counter2 value is " + c2.get());
        c1.reset();   // reset Counter c1
        System.out.println("Counter1 value is " + c1.get());
    }
}
Objects in memory

```
<table>
<thead>
<tr>
<th>Variables in main()</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td>c2</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>

class Counter

```

Data Hiding

- It is desirable to hide the inner details of a class (abstract data type) from the users of the class.
- We want to be able to determine the correctness of a class without examining the entire program of which it is a part.
- With our current class Counter we wish to assert that the value is always between 0 and 99.

Instance variables and methods

- As seen in the CounterTest example (and many previous examples using Strings), you invoke an instance method by expressions such as
  ```java
c1.click()
  ```
- You can use the same notation to access an instance variable. So in `main()` of CounterTest we could use
  ```java
c1.value
  ```

Accessing instance variables from outside the class breaks data hiding.

```java
class CounterTest2 {
    public static void main(String[] args) {
        Counter c1 = new Counter(); // create a Counter
        c1.value = 100; // breaks assumption about Counter
        System.out.println("Counter1 value is "+ c1.get());
    }
}
```

A better Counter.

```java
public class Counter {
    private int value; // instance variable
    public void reset() { value = 0; } // mutator method
    public int get() { return value; } // accessor method
    public void click() { value = (value + 1) % 100; }
}
```

public/private/default

- `private` methods/fields cannot be accessed from outside of the class
- `public` methods/fields can be accessed from anywhere
- default (no modifier) methods/fields have package access. They can be accessed from other classes in the same package.
  - If you don’t specify a package (see section 12.11), all classes in the same directory are part of the same, default - unnamed package.
Constructing objects

- Objects are created with
  
  ```java
  new ClassName()
  ```

- This allocates space for the object in the heap (memory), and initializes the object by invoking the constructor for the class if there is one.

- If there is no constructor, by default all fields are initialized (boolean fields are false, all other primitives are 0, and everything else is initialized to null).

Using constructors.

class CounterTest3 {
  public static void main(String[] args) {
    Counter c1 = new Counter(); // a Counter starting at 0
    Counter c2 = new Counter(50); // one starting at 50
    c1.click();
    c2.click();
    System.out.println("Counter1 value is " + c1.get());
    System.out.println("Counter2 value is " + c2.get());
  }
}

Adding constructors to Counter

```java
public class Counter {
  public Counter() { }
  public Counter(int v) { value = v % 100; }
  private int value; // instance variable
  public void reset() { value = 0; } // mutator method
  public int get() { return value; } // accessor method
  public void click() { value = (value + 1) % 100; }
}
```

Using constructors.

class CounterTest3 {
  public static void main(String[] args) {
    Counter c1 = new Counter(); // a Counter starting at 0
    Counter c2 = new Counter(50); // one starting at 50
    c1.click();
    c2.click();
    System.out.println("Counter1 value is " + c1.get());
    System.out.println("Counter2 value is " + c2.get());
  }
}

The default, no-arg constructor is only provided when there are no user specified constructors.

If we had added only the constructor

```java
public Counter(int v) { value = v % 100; }
```

then creating a Counter with

```java
Counter myCounter = new Counter();
```

would be a syntax error. There no longer is a constructor that takes zero arguments.

toString()

- toString() must be public. A full explanation must wait until Chapter 7. Every class has a default toString() that is public. When you give your class a toString() you can't undo the already public status of the method.

- By providing every class with a toString() method, we can use System.out.println() to print ANY object value.

class Person {
  int age;
  String name;
  char gender;
  public String toString() {
    return "Name: " + name + "\nAge: " + age + "\nGender: " + gender;
  }
}

class PersonTest {
  public static void main(String[] args) {
    ... System.out.println(jane);
    System.out.println(john);
  }
}
Static fields and methods

• Static methods don't operate (implicitly) on an instance of the class containing the method.
• Likewise, static fields are not part of an object, they are instead part of the class, hence also called class variables.

```java
public class Counter {
    private int value;
    private static int howMany = 0;
    public Counter() { howMany++; }
    public void reset() { value = 0; }
    public int get()    { return value; }
    public void click() { value = (value + 1) % 100; }
    public static int howMany() { return howMany; }
}
```

Adding a static method and a static field to Counter.

```java
// CounterTest2.java - demonstration of a static field
class CounterTest2 {
    public static void main(String[] args) {
        System.out.println(Counter.howMany());
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.click();
        c2.click();
        System.out.println("Counter1 value is "+c1.get()); //prints Counter1 value is 1
        System.out.println("Counter2 value is "+c2.get()); //prints Counter2 value is 2
        System.out.println(Counter.howMany()); // prints 2
    }
}
```

Objects in memory

```
c1
  value 1
  howMany 2
  reset()
  get()
  click()

value 2

c2

The Heap
```

System.out.println()

```
Class System
  PrintStream object
    Class PrintStream
      out
        println()
        print()
```

Calling Methods

• methods in the same class
  – just use the name
  – works for
    • instance method to instance method
    • instance method to static method
    • but NOT static method to instance method
• instance methods
  – objectReference.methodName()
• class methods
  – ClassName.methodName()
instance to instance

- We could implement `click()` in `Counter` with
  ```java
  void click() { value = (get() + 1) % 100; }
  ```
- This call to `get()` is operating on the same `Counter` object as the one used to invoke `click()`.

Why not static to instance?

- When calling one instance method in the same class from another in the same class, they both operate on the same, implicit object.
- When executing a static method there is NO implicit object being operated on, hence calling an instance method in the same class using only the method names, doesn’t specify what object to operate on.

When executing the call to `howMany()` below, what `Counter` object is being manipulated?

```java
// CounterTest2.java - demonstration of a static field
class CounterTest2 {
    public static void main(String[] args) {
        System.out.println(Counter.howMany()); // prints 2
    }
}
```

Answer: There isn't one. So trying to call `get()` from within `howMany()` like we did from within `click()` won’t work.

class `Change`

```java
private int dollars, quarters, dimes, pennies;
private double total;
Change(int dlrs, int qtr, int dm, int pen) {
    dollars = dlrs;
    quarters = qtr;
    dimes = dm;
    pennies = pen;
    total = dlrs + 0.25 * qtr + 0.1 * dm + 0.01 * pen;
}
```

```java
public String toString() {
    return (“$” + total + “
” + dollars + “ dollars
” + quarters + “ quarters
” + dimes + “ dimes
” + pennies + “ pennies
”);
}
```

class `ChangeTest`

```java
public class ChangeTest {
    public static void main(String[] args) {
        double owed = 12.37;
        double paid = 15.0;
        System.out.println(“You owe ” + owed);
        System.out.println(“You gave me ” + paid);
        System.out.println(“Your change is ” +
            Change.makeChange(15.0, 12.37));
    }
}
```

```java
static Change makeChange(double paid, double owed) {
    double diff = paid - owed;
    int dollars, quarters, dimes, pennies;
    dollars = (int)diff;
    pennies = (int)((diff - dollars) * 100);
    quarters = pennies / 25;
    pennies -= 25 * quarters;
    dimes = pennies / 10;
    pennies -= 10 * dimes;
    return new Change(dollars, quarters, dimes, pennies);
}
```
Change objects are immutable. You don't want someone modifying a change value, although you might create new values. How about an operation that actually operates on a Change value (other than converting it to a string)? Let's add two Change values.

```java
class Change {
    ...
    public Change add(Change addend) {
        Change result = new Change(dollars + addend.dollars,
                                quarters + addend.quarters,
                                dimes + addend.dimes,
                                pennies + addend.pennies);
        return result;
    }
}
```

An alternative method for adding together two Change values. This is NOT an OOP approach. It would require us to add the name of the class when calling the method,

```java
public class ChangeTest2 {
    public static void main(String[] args) {
        Change c1 = new Change(10, 3, 4, 3);
        Change c2 = new Change(7, 2, 2, 1);
        Change sum = c1.add(c2);
        // looks almost like  sum = c1 + c2;
        System.out.println(sum);
    }
}
```

Class method vs Instance method

An alternative method for adding together two Change values. This is NOT an OOP approach. It would require us to add the name of the class when calling the method,

```java
public class Change2 {
    public static Change add(Change augend, Change addend) {
        Change result =
                        new Change(augend.dollars + addend.dollars,
                                    augend.quarters + addend.quarters,
                                    augend.dimes + addend.dimes,
                                    augend.pennies + addend.pennies);
        return result;
    }
}
```

The NON-OOP way to do it.

```java
public class ChangeTest2 {
    public static void main(String[] args) {
        Change c1 = new Change(10, 3, 4, 3);
        Change c2 = new Change(7, 2, 2, 1);
        Change sum = Change.add(c1, c2);
        // looks almost like  sum = c1 + c2;
        System.out.println(sum);
    }
}
```

Scope

- The scope of class and instance variables is the entire class, regardless of where the declaration appears.
- Local variables can hide class-instance variables (have the same name). The local variable takes precedence. You can still access the class-instance variables with the keyword `this`.

```java
class Counter {
    void reset() { value = 0; }
    int get() { return value; }  // current value
    void click() { value = (value + 1) % 100; }
    int value;                    // 0 to 99
}
```

Notice that in the following, if you read from top to bottom, we use value before encountering the definition.
The scope of the class variable `x`, overlaps the scope of the local variable `x`.

```java
//Scope2.java: class versus local scope
class Scope2 {
    public static void main(String[] args) {
        int x = 2;
        System.out.println("local x = "+x);
        System.out.println("class variable x = "+Scope2.x);
    }
    static int x = 1;
}
```

class Change {
    private int dollars, quarters, dimes, pennies;
    ...
    Change(int dollars, int quarters, int dimes, int pennies)
    {
        this.dollars = dollars;
        this.quarters = quarters;
        this.dimes = dimes;
        this.pennies = pennies;
        total = dollars + 0.25 * quarters + 0.1 * dimes + pennies;
    }
    ...
}