Objects & Classes

Weiss ch. 3

- So far:
  - Point (see java.awt.Point)
  - String
  - Arrays of various kinds
  - IPAddress (see java.net.InetAddress)
- The Java API “library” has many more, organized into packages:
  - java.net: InetAddress, URL, etc.
  - javax.swing: Icon, JButton, JCheckBox, JWindow, etc.
  - java.lang: String, Compiler, etc.
  - java.io: File, FileInputStream, StreamTokenizer, etc.
  - *thousands more*

Objects Vs. Primitives

- Problem:
  - Every object in Java is an Object
  - An Object has an *equals* method (and a few others)
  - Primitive types are not objects
- Common scenario:

```
boolean allEquals(Object[ ] array) {
    Object first = array[0];
    for (int i = 1; i < array.length; i++) {
      if (! array[i].equals(first))
        return false;
    }
  }
```

- Works for objects of any type
- But not for primitive types (int, double, etc.)
  e.g. allEquals(new int[5]); → compiler error

Objects Vs. Primitives

- One (ugly) solution:

```
boolean allEquals(int[ ] array) {
    int first = array[0];
    for (int i = 1; i < array.length; i++) {
      if (array[i] != first)
        return false;
    }
  }
```

```
boolean allEquals(double[ ] array) {
    double first = array[0];
    for (int i = 1; i < array.length; i++) {
      if (array[i] != first)
        return false;
    }
  }
```

```
boolean allEquals(char[ ] array) {
    char first = array[0];
    for (int i = 1; i < array.length; i++) {
      if (array[i] != first)
        return false;
    }
  }
```

and 5 others!

Wrappers

- A better solution: wrap each primitive type in an object

```
Integer i = new Integer(5);
Integer j = new Integer(3);
boolean b = i.equals(j);
```

- Now we can use original code to compare:

```
Integer [ ] array;
Double [ ] array;
Character [ ] array;
```

- Also provides a convenient place to put methods relating to each primitive type:

  e.g., Integer.MAX_INT; Integer.parseInt( ), etc.

User Defined Types

- Objects are used by a program to encapsulate state and functionality
- Each object has a *type*
- Classes define the type of objects
  - specify per-object state and global state
  - specify per-object operations and global state
  - specify constructors to create new objects

Example Class

```
public class Date {
   private int month;
   private int day;
   private int year;

   public Date( ) {
      month = 1;
      day = 1;
      year = 1998;
   }

   public Date( int theMonth, int theDay, int theYear )
   {
      month = theMonth;
      day   = theDay;
      year  = theYear;
   }

   …
}
```

fields – one copy per Date instance

zero-argument constructor – initializes with defaults

three-argument constructor – initializes with given info

Instance Methods

```
public class Date {
   …

   // Conversion to String
   public String toString( )
   {
      return month + "/" + day + "/" + year;
   }

   public boolean equals( Object rhs ) {
      if( ! ( rhs instanceof Date ) )
         return false;
      Date rhDate = ( Date ) rhs;
      return rhDate.month == month &&
             rhDate.day == day &&
             rhDate.year == year;
   }

}
```

instance method: operates on an instance of Date

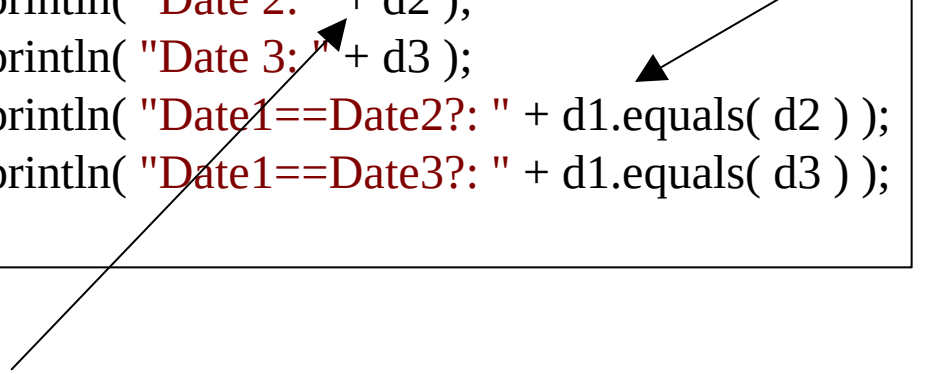
check to see if compared against another Date object

change an Object reference into a Date reference

Using Date

```
public static void main( String [ ] args )
  {
     Date d1 = new Date( );
     Date d2 = new Date( 1, 1, 1998 );
     Date d3 = new Date( 1, 1, 1999 );

     System.out.println( "Date 1: " + d1 );
     System.out.println( "Date 2: " + d2 );
     System.out.println( "Date 3: " + d3 );
     System.out.println( "Date1==Date2?: " + d1.equals( d2 ) );
     System.out.println( "Date1==Date3?: " + d1.equals( d3 ) );
  }
```



calling an instance method

String concatenation automatically calls Date.toString( )
This is equivalent to:

“Date 2: “ + d2.toString()

Getter and Setter Methods

```
public class Date {
   …
   // “getter” and “setter” methods
   public int getMonth( ) {
      return month;
   }

   public void setMonth(int m) {
      month = m;
   }

}
```

d1.setMonth(d1.getMonth() + 1)

Instance variable *month* is private.

Question: why bother?

this

```
public Date( int theMonth, int theDay, int theYear ) { … }
public void setMonth(int m) { …  }
```

- Often nice to use same name for parameter and for instance variable
- Often nice to have a reference to “the current object” inside an instance method

```
public Date( int month, int day, int year ) {
  this.month = month;
  this.day = day;
  this.year = year;
  System.out.println("Just constructed a new Date object: " + this);
 }
public void setMonth( int month) {
   if (month >= 1 && month <= 12)  this.month = month;
   else …
}
```

this

- Conceptually, dynamic calls:

```
public void setMonth(int month) { this.month = month; }
d1.setMonth(5)
```

are translated into:

```
public void setMonth(Date this, int month) { this.month = month; }
setMonth(d1, 5)
```

note: many languages do this literally
e.g. Python

Mixing Static and Instance Members

- Rule:

  static members → one global copy

  instance members → one copy per instance

accessing static variable

```
new Date(1,1,2004);
new Date(7,6,2004);
int i = Date.howMany( );
```

static “getter” method

```
public class Date {
  private static int count = 0;
  public Date( int month, int day, int year ) {
    count++;
    this.month = month; this.day = day; this.year = year;
}
  public static int howMany( ) { return count; }
}
```

- Only one *count* variable, shared among all Dates
- Initialized once, when class is loaded
- static methods may access *only* static variables

this as a constructor

- Bug in last example: forgot one Date constructor
  - Works: add the "count++" line to every constructor
  - Better: all defer to a single constructor, using *this*

```
public class Date {
  private static int count = 0;

  public Date( ) {
    this(1, 1, 1998);
  }

  public Date( int month, int day, int year ) {
    count++;
    this.month = month;
    this.day = day;
    this.year = year;
  }
  …
}
```

```
public class Date {
  private static int count = 0;

  public Date( ) {
     count++;
     month = 1; day = 1; year = 1998;
  }

  public Date( int month, int day, int year ) {
    this( );                    // begin with defaults
    this.month = month; // then modify
    this.day = day;
    this.year = year;
  }
  …
```

Note: either way, must be first line of constructor.