Chapter 9

Objects and Classes
Constructors

Fraction f = new Fraction();
Use a constructor to initialize an object

- A class may define as many constructors as needed
- Java provides one *default* constructor if you do not explicitly define a constructor
- A constructor has the same name as the class
First let’s look at the UML class Diagram for Distance.

```
Distance

- feetField: int
- inchesField: double
- labelField: String
- set(int feet, double inches, String label): void
+ getFeet(): int
+ setFeet(int feet): void
+ getInches(): double
+ setInches(double inches): void
+ getLabel(): String
```

Constructors are usually not shown in the UML class diagram; they are assumed.
public class Distance {
  private int feetField;
  private double inchesField;
  private String labelField;

  // This setter gets called by all other methods to perform
  // initialization. It is our designated setter.
  private void set(int feet, double inches, String label) {
    if (feet < 0) {
      System.out.println("Error: Negative feet::set to 0");
    } else {
      feetField = feet;
    } // end if
    if (inches < 0) {
      System.out.println("Error: Negative inches::set to 0");
    } else {
      inchesField = inches;
    } // end if
    labelField = label;
  } // end setDistance()

The reason the method set is private has to do with subclassing or inheritance, which is covered later in the book. Although we do not cover inheritance in this course, we should still follow proper practices from the get go. At least that way your code is ready, right out of the box.
public Distance() {
    this(0, 0.0, "na"); // call 3 argument constructor
    // set(0, 0.0, "na"); // or call set
} // end Distance()

// Designated constructor. All others will call this one.
public Distance(int feet, double inches, String label) {
    set(feet, inches, label);
} // end Distance()

public Distance(int feet, double inches) {
    this(feet, inches, "na");
    // set(feet, inches, "na");
} // end Distance()
public int getFeet() {
    return feetField;
}// end getFeet()

public void setFeet(int feet) {
    set(feet, inchesField, labelField);
}// end setFeet()
public double getInches() {
    return inchesField;
} // end getInches()

public void setInches(double inches) {
    set(feetField, inches, labelField);
} // end setInches()
@Override
public String toString() {
    return labelField + 
    ": " + 
    feetField + 
    ":" + 
    String.format("%.1f", inchesField);
} // end toString()
} // end Distance
package distancedemo;

public class DistanceDemo {
    public static void main(String[] args) {
        Distance d1 = new Distance();
        Distance d2 = new Distance(10, 6.5);
        Distance d3 = new Distance(10, 3.0, "d3");
        System.out.println(d1.toString());
        System.out.println(d2.toString());
        System.out.println(d3.toString());
    }
}
Application Deconstructed

<DistanceDemo.java>

```
run:
na: 0'-0.0"
ns: 10'-6.5"
dds: 10'-3.0"

BUILD SUCCESSFUL (total time: 0 seconds)
```
Application Deconstructed

<DistanceDemo.java>

d1.setDistance(20, -10.0, "d1");
System.out.println(d1.toString());
} // end main()
} // end DistanceDemo

Output - DistanceDemo [run]
run:
a: 0'-0.0"
a: 10'-6.5"
da: 10'-3.0"

Error: Negative inches::set to 0
d1: 20'-0.0"
BUILD SUCCESSFUL (total time: 1 second)
Recap

- Constructors have the same name as the class
- Java will provide a default constructor, if you do not provide any
- Constructors initialize the object
- All constructors should call the designated constructor
Q: Given the class `PointDemo`, which is **not** a valid constructor?

(A) `PointDemo()`  
(B) `PointDemo(int x, int y)`  
(C) `pointDemo(int x, int y)`  
(D) None of the above
Static Variables and Static Methods

static = shared
Let's examine main() more closely
When your application is loaded, a call is made to the method `main()`
main() is said to be the entry point of the application
Interestingly enough, the call is made without the use of an object
The magic is in the `static` keyword

```java
public static void main(String args[])
```
static makes the method a class-level method
Class-level or shared, means the method is called on the class, not an object.
package convertdemo;

public class Convert {
    private static final double DEGREES_PER_RADIAN = 180.0 / Math.PI;

    private static final double RADIANS_PER_DEGREE = Math.PI / 180.0;

    public static double toDegrees(double radians) {
        return radians * DEGREES_PER_RADIAN;
    }
    // end toDegrees()

    public static double toRadians(double degrees) {
        return degrees * RADIANS_PER_DEGREE;
    }
    // end toRadians()
public static double toCelcius(double fahrenheit) {
    return (fahrenheit - 32.0) / 1.8;
} // end toCelcius()

public static double toFahrenheit(double celcius) {
    return 1.8 * celcius + 32.0;
} // end toFahrenheit()
}// end Convert
package convertdemo;

public class ConvertDemo {

    public static void main(String[] args) {
        // Convert 45 degrees to radians.
        System.out.printf("45 degrees are equivalent to %1$.3f radians\n", Convert.toRadians(45.0));

        // Convert PI radians to degrees.
        System.out.printf("PI radians are equivalent to %1$.1f degrees\n", Convert.toDegrees(Math.PI));
    }

}
Application Deconstructed

<ConvertDemo.java>

    // Convert 0 degrees C to F.
    System.out.printf("0 degrees C are equivalent to %1$.1f
    degrees F\n", Convert.toFahrenheit(0.0));

    // Convert 212 degrees F to C.
    System.out.printf("212 degrees F are equivalent to %1$.1f
    degrees C\n", Convert.toCelcius(212.0));

}// end main()
}// end ConvertDemo

0 degrees C are equivalent to 32.0 degrees F
212 degrees F are equivalent to 100.0 degrees C
BUILD SUCCESSFUL (total time: 1 second)
Instance variables can also be shared by all objects

private static <type> <varName> = <initValue>
package deckdemo;

public class Deck {
    private static int cardsDealtSoFarField = 0;
    public static final int CARDS_IN_THE_DECK = 52;

    // Returns the number of cards dealt out so far.
    public static int cardsDealtCount() {
        return Deck.cardsDealtSoFarField;
    }
    // end cardsDealt()
}
// Displays a message indicating a card has been dealt.
public static void dealACard() {
    Deck.cardsDealtSoFarField += 1;
    if (Deck.cardsDealtCount() == 1) {
        System.out.println(Deck.cardsDealtCount() + " card was dealt.");
    } else {
        System.out.println(Deck.cardsDealtCount() + " cards were dealt.");
    }
}// end if
// Check if deck needs to be shuffled.
if (Deck.cardsDealtCount() == CARDS_IN_THE_DECK) {
    Deck.shuffle();
} // end if
} // end dealACard()

// Shuffles the deck by resetting the cardsDealtSoFarField to 0.
public static void shuffle() {
    System.out.println("... Shuffling the deck...");
    Deck.cardsDealtSoFarField = 0;
} // end shuffle()
} // end Deck
package deckdemo;

public class DeckDemo {

    public static void main(String[] args) {
        // Deal 52 cards.
        System.out.println("Cards dealt so far: " + Deck.cardsDealtCount());
        for (int card = 1; card <= Deck.CARDS_IN_THE_DECK; card++) {
            Deck.dealACard();
        } // end for
    }
} // end DeckDemo
// Deal two more cards.
Deck.dealACard();
Deck.dealACard();

// Reshuffle the deck and deal another card.
Deck.shuffle();
Deck.dealACard();
} // end main()
} // end DeckDemo

1 card was dealt.
2 cards were dealt.
... Shuffling the deck...
1 card was dealt.
BUILD SUCCESSFUL (total time: 1 s.
Recap

- **static** means shared by all objects (class-level)

- Instance methods require a receiving object

- **static** methods are called on the class, not a receiving object

- **static** methods cannot access instance level members (methods, variables) - no `this` passed
public void ....
Writing methods 101

- Try to write one method at a time
- Test each method immediately after coding it
- Use a driver (main() basically) to test each method
- Follow a bottom-up or top-down approach
Bottom-up tests
called methods first

```java
public void outputSquared(int value) {
    value = squareIt(value);
    System.out.println("Value squared: "+value);
}
```

caller method
this method will not be tested now; it doesn't even have to be implemented yet

```java
private int squareIt(int value) {
    return value * value;
}
```

called method
implement this method fully and call it directly from your driver
Top-down tests

caller first

```java
public void outputSquared(int value) {
    value = squareIt(value);
    System.out.println("Value squared: "+value);
}

private int squareIt(int value) {
    return 0;
}
```

Here we only interested in testing the caller. If we get the value (0) displayed or not

return any value that satisfies the compiler, since we are only testing the caller now
Recap

- Testing assures program is robust

- Test the caller first, by providing called stubs (syntactically proper only)

- Test the called first, by implementing fully and calling directly from driver
Develop a Time class and use both testing techniques.

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>- hourField: int               - 24 hour universal format</td>
</tr>
<tr>
<td>- minuteField: int</td>
</tr>
<tr>
<td>+ getHour(): int</td>
</tr>
<tr>
<td>+ setHour(int hour): void</td>
</tr>
<tr>
<td>+ getMinute(): int</td>
</tr>
<tr>
<td>+ setMinute(int minute): void</td>
</tr>
<tr>
<td>+ setTime(int hour, int minute): void</td>
</tr>
<tr>
<td>+ toString(): string</td>
</tr>
<tr>
<td>+ toStringStandard(): string  - 12 hour standard format</td>
</tr>
</tbody>
</table>
Overloading
A method can have many signatures

public Distance() {}  
public Distance(int feet, double inches) {}  
public Distance(int feet) {}  
public Distance(double inches) {}
A method's signature consists of its name and parameters
Overloading makes use of a method's signature
Overloaded methods have the same name
but vary in type and number of parameters
Java searches for closest method match

Distance d1 = new Distance();
Distance d2 = new Distance(10, 3.5);
Distance d3 = new Distance(10);
Distance d4 = new Distance(3.5);

public Distance() {}
public Distance(int feet, double inches) {}
public Distance(int feet) {}
public Distance(double inches) {}
Note: the return type is not part of the signature
Given these two methods

```java
int getValue()
double getValue()
```

What does Java do now?

I said: "These are the same; returns don't count"
Recap

- Methods can be overloaded
- Each method signature must be unique
- If methods differ only in their return type, Java will complain
Information
Hiding
Revisited
Object variables are references

- pc
- cpuField
- ramField
- pc object
- cpu object
- ram object
= only copies the reference

PC pc2 = pc1;

// Now, both pc1 and pc2 point to pc1's object
Objects can be changed from any reference

```java
pc2.setRam = "2 GB";

// This changes pc1's ram object as well.
// Both pc1 and pc2 refer to the same object after all.
```
Private object instance variables can be changed also

```java
Ram ram = pcl.getRam();
ram.setRam("4 GB");

// Although ramField is private in PC, this exposes
// it to changes.

// pcl's ram object is now "4 GB".
```
Recap

- Object variables as instance variables can cause subtle problems

- If you must protect instance objects, then return copies of them

- **Trade off**: Protection of object v. efficient memory use
Enumeration As A Class

```java
enum LightState {...}
```
enum is actually a class

```java
enum LightState {
    OFF, ON, DIMMED, FLICKERING
}
```

LightState class

public static objects
package enumdemo;

public class EnumDemo {
    enum LightState {
        // Each object is initialized to a color.
        OFF("black"),
        ON("white"),
        DIMMED("gray"),
        FLICKERING("red");

        private final String colorField;

        // Private final String colorField;

        // Private constructor to set the color.
        private LightState(String color) {
            colorField = color;
        }
        // end LightState()
    }
}
// Public accessor to get color.
public String getColor() {
    return colorField;
} // end getColor()

// end LightState

public static void main(String[] args) {
    LightState off = LightState.OFF;
    LightState on = LightState.ON;
    LightState dimmed = LightState.DIMMED;
    LightState flickering = LightState.FLICKERING;
System.out.println("State:" + off.toString() + " :: Color:" + off.getColor());
System.out.println("State:" + on.toString() + " :: Color:" + on.getColor());
System.out.println("State:" + dimmed.toString() + " :: Color:" + dimmed.getColor());
System.out.println("State:" + flickering.toString() + " :: Color:" + flickering.getColor());
}
}
// end EnumDemo
Recap

- An enum is actually a Java class
- You can use several methods to obtain enum functionality
- You can enhance the enum class with instance variables and methods
Packages
A package groups related classes together in a folder
The folder has the same name as the package
Each class within the package is in its own <className>.java file
Each class imports a class or all classes from the package
NetBeans specific: The name of package is under the src folder

package enumdemo
Recap

- A package groups related classes in a folder
- The folder has the same name as the package
- Import the class you want from the package it is in
- Packages organize your classes logically