Recall: Objects and Classes

- A class is a definition of a \textit{type of thing}
  - 1. State
  - 2. Behavior
- An object is a \textit{particular thing}
  - An object is an \textit{instance} of a class

Example : java.util.Random

```java
package java.util;
public class Random
    implements java.io.Serializable
{
    static final long serialVersionUID = 3905348978240129619L;
    private long seed;
    private final static long multiplier = 0x5DEECE66DL;
    private final static long addend = 0xBL;
    private final static long mask = (1L << 48) - 1;
    public Random() {...}
    public Random(long seed) {...}
    synchronized public void setSeed(long seed) {...}
    synchronized protected int next(int bits) {...}
    private static final int BITS_PER_BYTE = 8;
    private static final int BYTES_PER_INT = 4;
    public void nextBytes(byte[] bytes) {...}
    public int nextInt() {...}
    public int nextInt(int n) {...}
    public long nextLong() {...}
    public boolean nextBoolean() {...}
    public float nextFloat() {...}
    public double nextDouble() {...}
    private double nextNextGaussian;
    private boolean haveNextNextGaussian = false;
    synchronized public double nextGaussian() {...}
}
```

Constructors

- Constructors are special methods that get called with the \texttt{new} operator
  ```java
  Mobile m12 = new Mobile(10);;
  ```
- The name of a constructor is the same as the name of the class
- What does it do?
- What if there is no constructor?

Multiple Constructors

- There are often several constructors for any one class
- They all have the same name (the name of the class)
- They must differ in their parameter lists
  ```java
  Mobile m10 = new Mobile();;
  Mobile m12 = new Mobile(10);
  ```
- Return value?
Methods

- The method header declares the type and name for each required parameter

```java
/**
 * Add a simple weight right at the attachment point.
 * @param f a weight or lifting force
 */
public void addWeight(double f) {
    moby.add(new WeightAttachment(f));
}
```

Examples from class java.lang.String

- `toLowerCase()`
  Converts all of the characters in this String to lower case using the rules of the default locale

- `startsWith(String prefix)`
  Tests if this string starts with the specified prefix

- `substring(int beginIndex, int endIndex)`
  Returns a new string that is a substring of this string

Method calls

- Recall: `substring(int beginIndex, int endIndex)`

```java
int beginIndex = 1;
String myName = "Lex Luther";
String twoChar = myName.substring(beginIndex, beginIndex+2);
```

- If necessary and possible, the compiler will convert the value provided by the caller to the type of the value that was requested by the method in the formal parameter list

Returning a value to the caller

```java
/**
 * Get current X value.
 * @return the X coordinate
 */
public int getX() {
    return x;
}
```

“Accessor” methods

```java
public int getX()
public int getWidth()
```

Method Overloading

- Classes may declare multiple methods with the same name, provided the signature is different
- The signature of a method is:

```java
int beginIndex = 1;
String myName = "Lex Luther";
String twoChar = myName.substring(beginIndex, beginIndex+2);
```

- For example, `System.out.println` is overloaded for many types

  ```java
  println(char c);
  println(double d);
  println(String s);
  ```

Abstract the behavior of classes

- We sometimes want to use one or more methods that are available for various objects, even though they are not all of the same class
- Consider the attachments to a mobile
  - we want to know if it is balanced, what’s the weight, and what are the x and y torque values
- So we can promise that:
  - We don’t know exactly what kind of an attachment it is, but we do know that it can tell us if it is balanced, what the weight and torque values are
Interface

- You can say that any class that claims to be an *Attachment* will guarantee that it has methods for all the things that any *Attachment* must do.
- The definition of the interface shows exactly what the methods must look like.

```java
public interface Attachment {
    /**
     * Check to see if this Attachment is balanced.
     * An Attachment is balanced if all its constituent parts are balanced.
     * @return true if the Attachment is balanced, false if not.
     */
    public boolean isBalanced();

    /**
     * Return the total weight of this Attachment.
     * @return the total weight of this Attachment.
     */
    public double getWeight();

    /**
     * Get the x-torque. This is the torque applied around the x-axis by
     * the Attachment to the Mobile at the attachment point.
     * @return the torque around the x-axis.
     */
    public double getXTorque();

    /**
     * Get the y-torque. This is the torque applied around the y-axis by
     * the Attachment to the Mobile at the attachment point.
     * @return the torque around the y-axis.
     */
    public double getYTorque();
}
```

Using an interface in a class definition

- Each of the classes that wants to be considered an *Attachment* must say so at the very beginning of the class definition.

```java
public class Branch implements Attachment {
    // ...
}
```

Using Attachment Interface

```java
public class Mobile {
    Attachment attachments[] = ...;

    public double getWeight() {
        double w = 0;
        for (int ii=0; ii<attachments.length; ii++) {
            Attachment a = attachments[ii];
            w += a.getWeight();
        }
        return w;
    }
    // ...
}
```

Cast to Attachment

- Tell the compiler that the ArrayList contains objects that are *Attachments*.

```java
public class Mobile {
    ArrayList myList = ...;

    public double getWeight() {
        double w = 0;
        Iterator iter = myList.iterator();
        while (iter.hasNext()) {
            Attachment a = (Attachment)iter.next();
            w += a.getWeight();
        }
        return w;
    }
}
```

Relationships between classes

- Classes can be related via composition
  - This is often referred to as the “has-a” relationship
  - eg, a Mobile *has a* list in an ArrayList of Attachments
- Classes can also be related via inheritance
  - This is often referred to as the “is-a” relationship
  - eg, an ArrayList *is an* AbstractList
Mobile has a list of Attachments

Why use inheritance?

- Code simplification
  - Avoid doing the same operation in two places
  - Avoid storing “matching state” in two places
- Code simplification
  - We can deal with objects based on their common behavior, and don’t need to have special cases for each subtype
- Code simplification
  - Lots of elegant code has already been written - use it, don’t try to rewrite everything from scratch

Reduce the need for duplicated code

- Suppose we have two Attachments:
  - Branch has getXTorque() method
  - LiftingBranch has getXTorque() method
  - and they are implemented exactly the same way
- We can implement this method once in a base class, and then extend it and add or replace implementations of other methods as we like
LiftingBranch : subclass of Branch

```java
class LiftingBranch extends Branch {
    public LiftingBranch(double angle, double len, Mobile mA, Mobile mB) {
        this(angle, len, mA); // super() to call another constructor from same class
        lift = mB;
    }
}
```

```
Using the superclass constructor
• Constructor of the superclass is called to do much (or all) of the initialization for the subclass
```
**instanceof**

- Used to test an object for class membership
  
  ```java
  if (bunch.get(i) instanceof Branch) {...}
  ```

- Tests for a relationship anywhere along the hierarchy
  
  » Also tests whether an object’s class implements an interface

- What class must `<classname>` represent for the following expression to be true always?
  
  ```java
  if (v instanceof `<classname>`) {...}
  ```