More about classes
Logistics

• Homework 9 due tomorrow
• Will post Homework 10 over the next few days
  – Will be due two weeks from tomorrow
  – Last homework
• No class/recitation on Wednesday
Composition

• The most common way to use one class within another is composition—just have a variable of that type

• Examples:
  – class LunarLanderGame {
    LunarLander lander = new LunarLander();
    ...
  }
  – class MaxPlayer {
    String name;    // String is a class
    Game game;      // Game is a class
  }

• Composition is suitable when one class is composed of objects from another class, or needs frequent reference to objects of another class
Composition vs. Inheritance

• Inheritance is appropriate when one class is a *special case* of another class

• Example:
  – `class Animal { ... }`
  – `class Dog extends Animal { ... }`
  – `class Cat extends Animal { ... }`

• Use inheritance *only* when one class clearly specializes another class (and should have all the features of that superclass)

• Use composition in all other cases
Inheritance

class Animal {
    int row, column;                      // will be inherited
    private Model model;                // inherited but inaccessible
    Animal( ) { ... }                        // cannot be inherited
    void move(int direction) { ... } // will be inherited
}

class Rabbit extends Animal {
    // inherits row, column, move, but not constructor
    // model really is inherited, but you can’t access it
    int distanceToEdge;                 // new variable, not inherited
    int hideBehindBush( ) { ... }      // new method, not inherited
}
Assignment

• A member of a subclass is a member of the original class; a rabbit is an animal

    Animal animalBehindBush;
    Rabbit myRabbit;
    ...
    animalBehindBush = myRabbit; // perfectly legal

    myRabbit = animalBehindBush; // not legal

    myRabbit = (Rabbit)animalBehindBush;
    // legal syntax, but requires a runtime check
Arrays of Objects

• When you declare an array, you must specify the type of its elements:
  
  Animal animals[ ];

• However, **Object** is a type, so you can say:

  Object things[ ];       // declaration
  things = new Object[100];  // definition

  – You can put *any Object* in this array:

    things[0] = new Fox();
Wrappers

- Each kind of primitive has a corresponding wrapper (or envelope) object:
  - byte     Byte
  - short    Short
  - int      Integer (*not Int*)
  - long     Long
  - char     Character (*not Char*)
  - boolean  Boolean
  - float    Float
  - double   Double
Wrapper constructors

- Each kind of **wrapper** has at least one constructor:
  - Byte byteWrapper = new Byte(byte value)
  - Short shortWrapper = new Short(short value)
  - Integer intWrapper = new Integer(int value)
  - Long longWrapper = new Long(long value)
  - Character charWrapper = new Character(char value)
  - Boolean booleanWrapper = new Boolean(boolean value)
  - Float floatWrapper = new Float(float value)
  - Double doubleWrapper = new Double(double value)
Some wrapper methods

- Wrapper classes have other interesting features
  - variables:
    - `Integer.MAX_VALUE = 2147483647`
  - methods:
    - `Integer.toHexString(number)`
    - `anyType.toString();`
Why bother with wrappers?

Prior to Java 5, you *couldn’t* do this:
```java
things[1] = 5;
```

But you *could* do this:
```java
things[1] = new Integer(5);
```

You *couldn’t* do this:
```java
int number = things[1];
```

But you *could* do this:
```java
int number = ((Integer)things[1]).intValue();
```
Types and values

• A variable has both a type and a value
• Consider Animal animal;
  – The type of variable animal is Animal
    • The type of a variable never changes
    • The syntax checker can only know about the type
  – The value of animal might sometimes be a rabbit and at other times be a fox
    • Messages such as animal.run() are sent to the value
    • The value (object) determines which method to use
Calling methods

• Java must ensure that every method call is legal
  – That is, the object receiving the message must have a corresponding method
• But when the Java compiler checks syntax, it can’t know what the value of a variable will be; it has to depend on the type of the variable
  – If the variable is of type T, then either
    • Class T must define an appropriate method, or
    • Class T must inherit an appropriate method from a superclass, or
    • Class T must implement an interface that declares an appropriate method
Overriding methods

```java
class Animal {
    int decideMove( ) {
        return Model.STAY;
    }
}

class Rabbit extends Animal {
    // override decideMove
    int decideMove( ) { // same signature
        return random(Model.MIN_DIRECTION,
                       Model.MAX_DIRECTION);
    }
}
```
Overriding methods II

• When you override a method:
  – You must have the exact same signature
  – Otherwise you are just *overloading* the method, and both versions of the method are available

• When you override a method, you cannot make it more private
  – In this example, *Animal* defines a method
  – *Every* subclass of *Animal* *must* inherit that method, including subclasses of subclasses
  – Making a method more private would defeat inheritance
Some methods cannot be overridden

```java
class Animal {
    final boolean canMove(int direction) { ... }
}

class Rabbit extends Animal {
    // inherits but cannot override canMove(int)
}
```
Some variables cannot be redefined

- class BorderLayout {
  public static final String NORTH = "North";
}

- If you were to create a subclass of BorderLayout, you would not be able to redefine NORTH

- Static instance variables are the same for the entire class, i.e., for all objects
  - Accessed using the class name, e.g., BorderLayout.NORTH
  - Usually you don’t want to change them so static often goes with final
  - Also possible to change, e.g., to count how many instances you have
Some classes cannot be extended

- final class `StringContent` { ... }

- When an entire class is made `final`, it cannot be extended (subclassed)
- Making a class `final` allows some extra optimizations
- Very few Java-supplied classes are `final`

- `final` classes are a bad idea in general – programs almost always get used in ways that were not foreseen by their authors
Some objects cannot be altered

• An **immutable** object is one that cannot be changed once it has been created
• **Strings** are immutable objects
• It’s easy to make an object immutable:
  – Make all its fields **private**
  – Provide *no* methods that change the object
Extend, don’t modify

- **The Open-Closed Principle**: Software entities (classes, modules, methods, and so forth) should be *open for extension* but *closed for modification*
  - You should design classes that can be extended
  - You should *never* have to modify a class in order to extend it
  - If you *do* have to change code, your superclass was poorly designed