More about Classes

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 1:
  - class Animal { … }
  - class Dog extends Animal { … }
  - class Cat extends Animal { … }
- Example 2:
  - class Player { … }
  - class ComputerPlayer extends Player { … }
  - class HumanPlayer extends Player { … }
- 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
```

Assignment II

animalBehindBush = myRabbit; is legal--but *why*?

```
int NUMBER_OF_ANIMALS = 8;
Animal animals[ ] = new Animal[NUMBER_OF_ANIMALS];
animals[0] = new Rabbit();
animals[1] = new Seagull();
animals[2] = new Snail();
...
for (int i = 0; i < NUMBER_OF_ANIMALS; i++)
    animals[i].move(); // legal if defined in Animal
```

Assignment III

- From previous slide:

```
for (int i = 0; i < NUMBER_OR_ANIMALS; i++)
    animals[i].allowMove(); // legal if defined in Animal
```

- But:

```
for (int i = 0; i < NUMBER_OR_ANIMALS; i++) {
    if (animals[i] instanceof Rabbit) {
        ((Rabbit)animals[i]).tryToHide();
    }
}
```

- Here, tryToHide() is defined only for rabbits
  - We must check whether animals[i] is a rabbit
  - We must *cast* animals[i] to Rabbit before Java will allow us to call a method that does not apply to *all* Animals
  - After the if test, you might think Java "knows" that animals[i] is a Rabbit—but it doesn't

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();
    ```

  - But (before Java 5) you *cannot* do this:

    ```
    things[1] = 5;                // why not?
    ```

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***)

More wrapper constructors

- Every wrapper type *except* Character has a constructor that takes a String as an argument
  - Example: Double d = new Double("3.1416");
  - Example: Boolean b = new Boolean("true");
- The constructors for the numeric types can throw a NumberFormatException:
  - Example: Integer i = new Integer("Hello");

Wrapper “deconstructors”

- You can retrieve the values from wrapper objects:
  - byte by = byteWrapper.byteValue();
  - short s = shortWrapper.shortValue();
  - int i = intWrapper.intValue();
  - long l = longWrapper.longValue();
  - char c = charWrapper.charValue();
  - boolean bo = booleanWrapper.booleanValue();
  - float f = floatWrapper.floatValue();
  - double d = doubleWrapper.doubleValue();

Additional wrapper methods

- Wrapper classes have other interesting features
  - variables:
    - Integer.MAX_VALUE = 2147483647
  - methods:
    - Integer.toHexString(number)
    - anyType.toString();

Back to arrays

- Why bother with wrappers?
- `Object[ ] things = new Object[100];`
- Prior to Java 5, you *cannot* do this:

```
things[1] = 5;
```

- But you *could* do this:

```
things[1] = new Integer(5);
```

- You *couldn't* do this:

```
int number = things[1];
```

- But you *could* do this:

```
int number = ((Integer)things[1]).intValue();
```

Auto-boxing and auto-unboxing

- Since version 5, Java will automatically box (wrap) primitives when necessary, and unbox (unwrap) wrapped primitives when necessary
- You can now do the following (where things is an array of Object) :
  - things[1] = 5; instead of
    things[1] = new Integer(5);
  - int number = (Integer)things[1]; instead of
    int number = ((Integer)things[1]).intValue();
    but not int number = (int)things[1];

equals and other methods

- Some methods, such as equals, take an Object parameter
  - Example: if (myString.equals("abc")) { … }
- JUnit's assertEquals(*expected*, *actual*) also takes objects as arguments
- Auto boxing and unboxing, while convenient, can lead to some strange problems:

  -
    ```
    Integer foo = new Integer(5);
    Integer bar = new Integer(5);
    ```
  - Now: foo == 5 is true
    bar == 5 is true
    foo.equals(bar) is true
    foo.equals(5) is true
    foo == bar is false

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

Sending messages

- Java must ensure that every message 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

```
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

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

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

Some variables cannot be shadowed

-
  ```
  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

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 classes cannot be instantiated

- In the AWT, TextField extends TextComponent, and TextComponent extends Component
- You can create (instantiate) a TextField, but you cannot directly create either a TextComponent or a Component
- What is it that prevents you from doing so?
  - Component is an abstract class, and abstract classes cannot be instantiated
  - TextComponent has an explicit constructor, but it is private

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
  - Provide *no* methods that return a mutable

You can always be more specific

- Rule: Design subclasses so they may be used anywhere their superclasses may be used
  - If a Rabbit is an Animal, you should be able to use a Rabbit object anywhere that an Animal object is expected

Don’t change the superclass

- The Liskov Substitution Principle: Methods that use references to base classes must be able to use objects of derived classes without knowing it
  - If you introduce a `Deer` class, you should not have to make any changes to code that uses an `Animal`
  - If you *do* have to change code, your `Animal` class was poorly designed

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

Related style rules, I

- Rule: Define small classes and small methods.
  - Smaller classes and methods are easier to write, understand, debug, and use
  - Smaller classes are more focused--they do only one thing
    - This makes them easier to extend

Related style rules, II

- Rule: Build classes from primitives and Java-defined classes; avoid dependence on program-specific classes
  - The less your class depends on others, the less it has to be “fixed” when the others change
  - If your class is stand-alone, maybe it can be used in some future program

Related style rules, III

- Rule: Make all fields private.
  - Private fields are controlled by your class; no other class can snoop at them or meddle with them
  - This means you can change them if necessary
  - You can provide setter and getter methods (to set and get field values) *when you think it is appropriate* to give this kind of access
  - Even if you provide setter and getter methods, you maintain a measure of control

Related style rules, IV

- Rule: Use polymorphism instead of instanceof
  - Bad:

```
class Animal {
    void move() {
        if (this instanceof Rabbit) { … }
        else if (this instanceof Fox) { … }
}   }
```

  - Good:

```
class Rabbit extends Animal {
    void move() { … }
}
class Fox extends Animal {
    void move() { … }
}
```

The End