Advanced Object-oriented Programming

Lecture 5

Modifiers
Battle of the Bands

Recall:

class BandCard constructor

BandCard(String n,
            int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;
}
When a BandCard instance is created, its fields are \textit{initialised} in the constructor. Thereafter, the values of the field (\textit{should}) never change.

In other words, the fields should be \texttt{final}:

```java
class BandCard
{
    final int volume;
}
```

(similarly for all the other fields).

\texttt{final} variables can not be assigned to
(unless they’re fields, and then \textit{only in a constructor}).
When a BandCard instance is created, its fields are *initialised* in the constructor. Thereafter, the values of the field (*should*) never change.

In other words, the fields should be *final*:

```java
final int volume;
```

(similarly for all the other fields).

*final* variables cannot be assigned to (unless they’re fields, and then *only in a constructor*).
Modifier: final

When a BandCard instance is created, its fields are *initialised* in the constructor. Thereafter, the values of the field (*should*) never change.

In other words, the fields should be **final**:

```
class BandCard
{
  final int volume;
}
```

(similarly for all the other fields).

**final** variables can not be assigned to (unless they’re fields, and then *only in a constructor*).
Any attempt to assign to a `final` variable will cause a compile-time error:

```java
class FinalExample {
    final int fin;

    FinalExample(int val) {
        fin = val; // okay in the constructor
    }

    void setFin(int val) {
        fin = val; // compile-time error
    }
}
```
Any attempt to assign to a `final` variable will cause a compile-time error:

```java
class FinalExample {
    final int fin;
    FinalExample(int val) {
        fin = val;  // okay in the constructor
    }
    void setFin(int val) {
        fin = val;  // compile-time error
    }
}
```
Any attempt to assign to a `final` variable will cause a compile-time error:

class FinalExample {
    final int fin;

    FinalExample(int val) {
        fin = val; // okay in the constructor
    }

    void setFin(int val) {
        fin = val; // compile-time error
    }
}

Modifier: final
Modifier: final

Any attempt to assign to a `final` variable will cause a compile-time error:

```java
class FinalExample {
    final int fin;

    FinalExample(int val) {
        fin = val; // okay in the constructor
    }

    void setFin(int val) {
        fin = val; // compile-time error
    }
}
```
Reading Compiler Error Messages

When you compile the code above, you get this error message:

```
 uh-oh: terminal output

FinalExample.java:13: cannot assign a value to final variable fin
fin = val;
```

The file where the error occurs
The line number where the error occurs
What the error is
The offending piece of code (line 13 of FinalExample.java)
Reading Compiler Error Messages

When you compile the code above, you get this error message:

| FinalExample.java:13: cannot assign a value to final variable fin |
| fin = val; |

The file where the error occurs
The line number where the error occurs
What the error is
The offending piece of code (line 13 of FinalExample.java)
Reading Compiler Error Messages

When you compile the code above, you get this error message:

```
uh-oh: terminal output

FinalExample.java:13: cannot assign a value to final variable fin
  fin = val;
```

The file where the error occurs
The line number where the error occurs
What the error is
The offending piece of code (line 13 of FinalExample.java)
When you compile the code above, you get this error message:

```
uh-oh: terminal output
FinalExample.java:13: cannot assign a value to final variable fin
fin = val;
```

The file where the error occurs
The line number where the error occurs
What the error is
The offending piece of code (line 13 of FinalExample.java)
Reading Compiler Error Messages

When you compile the code above, you get this error message:

```
FinialExample.java:13: cannot assign a value to final variable fin
fin = val;
```

The file where the error occurs: `FinalExample.java`
The line number where the error occurs: 13
What the error is: Cannot assign a value to a final variable.
The offending piece of code (line 13 of FinalExample.java):
```
fin = val;
```
Why Say What File Contains the Error?

The job of the compiler is to produce byte-code for every method in every class in a given file:

For example

```
javac BandCard.java
```

compiles all the `BandCard` methods — i.e., all the code in `Bandcard.java`

(but there's no main method in that file)

We need more classes to achieve our aim of implementing Battle of the Bands.
Why Say What File Contains the Error?

The job of the compiler is to produce byte-code for every method in every class in a given file:

For example

```
javac BandCard.java
```

does not compile all the `BandCard` methods — i.e., all the code in `Bandcard.java`
(but there’s no main method in that file)

We need more classes to achieve our aim of implementing Battle of the Bands.
Why Say What File Contains the Error?

The job of the compiler is to produce byte-code for every method in every class in a given file:

For example

```terminal
javac BandCard.java
```

compiles all the BandCard methods — i.e., all the code in Bandcard.java (but there’s no main method in that file)

We need more classes to achieve our aim of implementing Battle of the Bands.
Why Say What File Contains the Error?

The job of the compiler is to produce byte-code for every method in every class in a given file:

For example

```java
javac BandCard.java
```

compiles all the BandCard methods — i.e., all the code in Bandcard.java
(but there’s no main method in that file)

We need more classes to achieve our aim of implementing Battle of the Bands.
/** Pack of cards for Battle of the Bands. */

class CardPack {

    BandCard [] thePack = new BandCard[60];

    void fillThePack() {
        thePack[0] = new BandCard(...);
        thePack[1] = new Bandcard(...);
        ...
    } 

    ...
}
/** Pack of cards for Battle of the Bands. */

class CardPack {

    BandCard [] thePack = new BandCard[60];

    void fillThePack() {
        thePack[0] = new BandCard(...);
        thePack[1] = new Bandcard(...);
        ...
    }

    ...
}
Some More Classes

```java
/** Pack of cards for Battle of the Bands. */
class CardPack {
    BandCard [] thePack = new BandCard[60];
    void fillThePack() {
        thePack[0] = new BandCard(...);
        thePack[1] = new BandCard(...);
        ...
    }
    ...
}
```
Some More Classes

```java
/** Hand of cards for Battle of the Bands player. */

class CardHand {
    BandCard[] myCards = new BandCard[60];

    /** Add a card to the hand. */
    public void add(BandCard b) {
        ...
    }

    ...
}
```

A ‘magic number’ — we’ll talk about this later
Some More Classes

```java
/** Hand of cards for Battle of the Bands player. */
class CardHand {
    BandCard[] myCards = new BandCard[60];

    /** Add a card to the hand. */
    public void add(BandCard b) {
        ...
    }
}
```

A ‘magic number’ — we’ll talk about this later
/** Hand of cards for Battle of the Bands player. */
class CardHand {
    BandCard[] myCards = new BandCard[60];
    /** Add a card to the hand. */
    public void add(BandCard b) {
        ...
    }
    ...
}

A ‘magic number’ — we’ll talk about this later
Some More Classes

```java
/** Hand of cards for Battle of the Bands player. */
class CardHand {
    BandCard[] myCards = new BandCard[60];
    /** Add a card to the hand. */
    public void add(BandCard b) {
        ...
    }
    ...
}
```

A ‘magic number’ — we’ll talk about this later
A Top-level Application

pseudo code for file BattleOfTheBands.java

// create a CardPack
// create two CardHands, one for each player
// use add(BandCard) to fill each hand
// play

Pseudocode: informal description that is almost like code (sequences of instructions)
A Top-level Application

pseudocode for file BattleOfTheBands.java

// create a CardPack

// create two CardHands, one for each player

// use add(BandCard) to fill each hand

// play

Pseudocode: informal description that is almost like code (sequences of instructions)
A Top-level Application

pseudocode for file BattleOfTheBands.java

// create a CardPack
// create two CardHands, one for each player
// use add(BandCard) to fill each hand
// play

Pseudocode: informal description that is almost like code (sequences of instructions)
A Top-level Application

**pseudocode** for file BattleOfTheBands.java

```java
// create a CardPack

// create two CardHands, one for each player

// use add(BandCard) to fill each hand

// play
```

Pseudocode: informal description that is almost like code (sequences of instructions)
Compiling Multiple Classes

Classes **CardPack** and **CardHand** both use class **BandPack**.

Compiling one file, for example:

```
 javac CardPack.java
```

requires the compiler to use compiled code from class **BandCard**
(for example, to call the **BandCard** constructor)

If the file Bandcard.class does not exist, the compiler will look for file BandCard.java, and compile it.

So compiling one file may cause other files to be compiled.
Compiling Multiple Classes

Classes CardPack and CardHand both use class BandPack. Compiling one file, for example:

```
javac CardPack.java
```

requires the compiler to use compiled code from class BandCard (for example, to call the BandCard constructor)

If the file Bandcard.class does not exist, the compiler will look for file BandCard.java, and compile it.

So compiling one file may cause other files to be compiled.
Compiling Multiple Classes

Classes `CardPack` and `CardHand` both use class `BandPack`. Compiling one file, for example:

```
javac CardPack.java
```

requires the compiler to use compiled code from class `BandCard` (for example, to call the `BandCard` constructor)

If the file `Bandcard.class` does not exist, the compiler will look for file `BandCard.java`, and compile it.

So compiling one file may cause other files to be compiled.
Back to the Battling Bands

Our class BandCard now looks like:

```java
class BandCard {
    final String name;
    final int volume;
    final int attitude;
    final int cool;
    final int eclecticism;
    final int hair;

    ... (unchanged)
}
```

Remember we *can* assign to final variables inside the constructor — but only there.
Back to the Battling Bands

Our class **BandCard** now looks like:

```java
class BandCard {
    final String name;
    final int volume;
    final int attitude;
    final int cool;
    final int eclecticism;
    final int hair;

    ... (unchanged)
}
```

Remember we *can* assign to **final** variables inside the constructor — but only there.
This prevents ‘cheating’ code:

```java
BandCard myCard = ...;
BandCard opponentsCard = ...;
opponentsCard.volume = 0;  // cheat!
if (myCard.getVolume() >
    opponentsCard.getVolume())
System.out.println("I win");
```
A Beneficial Side-effect

This prevents ‘cheating’ code:

```java
BandCard myCard = ...;
BandCard opponentsCard = ...;
opponentsCard.volume = 0; // cheat!
if (myCard.getVolume() >
    opponentsCard.getVolume())
    System.out.println("I win");
```
Why Final Declarations

The main benefits of using `final` are:

- **Documentation**: it makes explicit the programmer’s intention that the value of a variable will not change.

- **Efficiency**: compiled code can sometimes be optimised to run faster. For example, a reference to a final variable can simply be replaced by its value at compile-time, rather than looking up the value at run-time.
Why Final Declarations

The main benefits of using `final` are:

- **Documentation**: it makes explicit the programmer’s intention that the value of a variable will not change.

- **Efficiency**: compiled code can sometimes be optimised to run faster. For example, a reference to a final variable can simply be replaced by its value at compile-time, rather than looking up the value at run-time.
CONSTANTS: final static

The modifier `final` is often combined with `static` to declare a constant. (By convention, names of constants are in ALL-CAPITALS.)

Constants are ‘variables’ whose value is fixed (`final`), and which are independent of instances (`static`).

Constants can therefore be accessed, not through instances, but through the class name: `Class.CONSTANT`.
For Example

Suppose volume values for all bands are meant to be in the range 0..11.

```java
class BandCard {
    static final int MAXIMUM_VOLUME = 11;
    ...
}
...
if (someValue > BandCard.MAXIMUM_VOLUME) {
    ...
}
```

The name of the variable
Static variable, so ‘dot’ notation uses the class name
For Example

Suppose volume values for all bands are meant to be in the range 0..11.

```java
class BandCard {
    static final int MAXIMUM_VOLUME = 11;
    ...
}
...
if (someValue > BandCard.MAXIMUM_VOLUME) {
    ...
}
```

The name of the variable

Static variable, so ‘dot’ notation uses the class name
For Example

Suppose volume values for all bands are meant to be in the range 0..11.

```java
class BandCard {
    static final int MAXIMUM_VOLUME = 11;
    ...
}
...
if (someValue > BandCard.MAXIMUM_VOLUME) {
    ...
}
```

The name of the variable

Static variable, so ‘dot’ notation uses the class name
For Example

Suppose volume values for all bands are meant to be in the range 0..11.

class BandCard {
    static final int MAXIMUM_VOLUME = 11;
    ...
}
...
if (someValue > BandCard.MAXIMUM_VOLUME) {
    ...
}

The name of the variable
Static variable, so ‘dot’ notation uses the class name
Suppose volume values for all bands are meant to be in the range 0..11.

```java
class BandCard {
    static final int MAXIMUM_VOLUME = 11;
    ...
}
...
if (someValue > BandCard.MAXIMUM_VOLUME) {
    ...
}
```

The name of the variable
**Static** variable, so ‘dot’ notation uses the **class name**
Static fields are often called **class fields** (as opposed to **instance** fields).

They are stored *once* in memory, rather than once for each instance

(cf. the \texttt{xCoord} instance fields in class \texttt{Point}, where each instance had its own \texttt{xCoord} field).
Storing in a unique memory location makes sense for constants (no point in storing more than one copy of a constant).

It also makes sense for functions that only depend upon their parameters, for example,

- `Math.sin(double)`
- `Math.random(int)`

In fact, class `Math` is largely a repository of static constants and methods

**Exercise**: add static modifiers to `CardPack` and `CardHand`
Storing in a unique memory location makes sense for constants (no point in storing more than one copy of a constant).

It also makes sense for functions that only depend upon their parameters, for example,

- `Math.sin(double)`
- `Math.random(int)`

In fact, class `Math` is largely a repository of static constants and methods.

**Exercise:** add `static` modifiers to `CardPack` and `CardHand`
More Static Fields

Class fields can be useful for computing dynamic data on classes.

For example, suppose we wanted to keep track of the *actual* maximum volume value of the band cards that have been created.

Obviously, this is not a property of the class itself, but will depend on how it is used: i.e., it depends on the volume values of the instances that have been created in a program.

class BandCard {
    static int greatestVolume = 0;
    final String name;
    ...
}
More Static Fields

Class fields can be useful for computing dynamic data on classes.

For example, suppose we wanted to keep track of the *actual* maximum volume value of the band cards that have been created.

Obviously, this is not a property of the class itself, but will depend on how it is used:

i.e., it depends on the volume values of the instances that have been created in a program.

class BandCard {
    static int greatestVolume = 0;
    final String name;
    ...
}
More Static Fields

Class fields can be useful for computing dynamic data on classes.

For example, suppose we wanted to keep track of the *actual* maximum volume value of the band cards that have been created.

Obviously, this is not a property of the class itself, but will depend on how it is used: i.e., it depends on the volume values of the instances that have been created in a program.

class BandCard {
    static int greatestVolume = 0;
    final String name;
    ...
}
class BandCard constructor

BandCard(String s,
        int v, int a, int c, int e, int h) {
    name = n;
    volume = v;
    attitude = a;
    cool = c;
    eclecticism = e;
    hair = h;

    if (volume > greatestVolume) {
        greatestVolume = volume;
    }
}
More Static Fields

```java
BandCard bc;
bc = new BandCard("a", 3, 0, 0, 0, 0);
b = new BandCard("b", 8, 0, 0, 0, 0);
b = new BandCard("c", 7, 0, 0, 0, 0);
System.out.println("greatest volume is: " +
greatestVolume);
```

We should get

```
greatest volume is: 8
```
More Static Fields

```java
example main method

BandCard bc;
bc = new BandCard("a", 3, 0, 0, 0, 0);
bc = new BandCard("b", 8, 0, 0, 0, 0);
bc = new BandCard("c", 7, 0, 0, 0, 0);
System.out.println("greatest volume is: " +
greatestVolume);
```

We should get

terminal output

greatest volume is: 8
That’s All, Folks!

Summary

- final (constant)
- static (class)

Next:
Scope