CSC 325 Adv Data Structures

Lecture 4

The Final Keyword

This is another misunderstood keyword, or at least a keyword that's not completely understood. This keyword can be used on fields, local variables, methods, and even classes. First let's go over the uses you should already be familiar with (so this part should mostly be review). Using 'final' on a field or local variable is nothing new. However, it may surprise you that there is more to the story than what you have been told.

First, recall that a variable marked as 'final' cannot change its value. This is actually great for compiler optimization. If the compiler knows that the variable never changes value, the type of byte code it produces can be better optimized and will utilize CPU registers in a more efficient fashion. Let's look at an example of its use and where you can initialize it based on the type of variable it is applied to:

```
public class Conversions {
  // constant class variables
  private static final int INCHES_TO_FEET = 12;
  private static final int INCHES_TO_YARD;

  // constant instance variables
  private final float OUNCES_IN_A_CUP = 8;
  private final boolean USES_METRIC;

  // static block
  static {
    INCHES_TO_YARD = 36;
  }

  // constructor
  public Conversions(boolean usesMetric) {
    USES_METRIC = usesMetric;
  }

  // method
  public float milesToKilometers(float numMiles) {
    // constant local variable
    final float MILES_IN_A_KM = 1.60934f;

    return numMiles * MILES_IN_A_KM;
  }
}
```

If a field is marked as final and is static (like INCHES_TO_FEET), then you have two options: 1. initialize it during declaration (like INCHES_TO_FEET) or 2. in a static block (like INCHES_TO_YARD). If a field is marked as final and is *not* static, then you, again, have two options: 1. initialize it during declaration (like OUNCES_IN_A_CUP) or initialize it in the constructor (like USES_METRIC). If a local variable is marked as final (like MILES_IN_A_KM) then initialization either happens at declaration or sometime later, as long as you don't assign to it after it

receives its initial value. You must follow these rules on where to initialize your final variables or Java will report an error. To recap:

Type of variable	Where to initialize
static field	at declaration or inside static block
non-static field	at declaration or inside constructor
local	at declaration or some point later in the function

If you try to overwrite a final variable at any point in the code, an error will be produced. Even if that code wouldn't technically be executed, the compiler will still complain:

```
class Apple {
  public final String size;

  public Apple() {
    size = "large";
  }

  public void eat() {
    size = "large with bite taken out";
  }
}

public class Program {
  public static void main(String[] args) {
    final int counter = 0;
    if (counter > 0) {
      counter++;
    }

    Apple a = new Apple();
  }
}
```

```
$ javac Program.java
Program.java:9: error: cannot assign a value to final variable size
    size = "large with bite taken out";
    ^
Program.java:17: error: cannot assign a value to final variable counter
      counter++;
      ^
2 errors
```

As you can see, we never actually call the "eat" method nor would we execute "counter++" since counter is not greater than 0. I personally agree with the compiler's actions here as this is a bomb waiting to go off. Just because you aren't pulling the pin right now, doesn't mean someone else won't come along to pull it later (e.g. one might add the line "a.eat()" at a later time).

Runtime and Compile-time constants

Knowing that a variable holds a constant value, some languages will completely remove the variable and just replace its usage with the value it holds (like a find and replace). This happens during compile time and the resulting low level code (i.e. assembly or byte code) is faster since it doesn't have to go to memory for the value of the variable (as the variable is no longer there, just its value). We call this a compile-time constant. In Java, if a final variable is of a primitive type or a String, Java will treat these as compile-time constants. But what about if the final variable is something else, like a reference type (other than String)? Let's look at the following code to answer this question:

```
class Vector2 {
  public int x;
  public int y;

  public Vector2(int x, int y) {
    this.x = x;
    this.y = y;
  }
}

class Line {
  public final Vector2 pt1;
  public final Vector2 pt2;

  public Line(int x1, int y1, int x2, int y2) {
    pt1 = new Vector2(x1, y1);
    pt2 = new Vector2(x2, y2);
  }
}

public class Program {
  public static void main(String[] args) {
    Line line1 = new Line(0, 0, 1, 1);
    Line line2 = new Line(0, 1, 2, 3);
  }
}
```

After you create a Line object, you aren't allowed to move the point locations (since they are marked as final). Can we simply replace pt1 and pt2 with whatever their constant value is (i.e. can we treat them as compile-time constants)? No, because their value is different for different Line objects. Ok, so could we replace their respective values for line1 and for line2? No, because line1 and line2 aren't created until we actually execute the program. This leads us to believe that not all final variables can be compile-time constants.

The "final" keyword can actually be used for either a compile time constant or a runtime constant. With a run time constant, the value of the variable is only evaluated during runtime. The compiler's job is not to swap out the variable, but instead it is to ensure it can never be overwritten, so that the runtime can properly do its job when it encounters this. Languages like C++ also allows both runtime and compile time constants. If you use the "const" keyword on a variable and assign it a constant value (like a literal), then it is treated as a compile

time constant and fully optimized. If you give a constant variable the value of a non-constant variable, it is treated as a runtime constant instead (and so optimization will not be as good). This is because non-constant variables can change multiple times during execution and so replacing the constant variable with its value is impossible at compile time.

```
#include <iostream>
using namespace std;

int main() {
  const int counter = 54;

  int var1 = 3;
  const int var2 = var1;
}
```

Here we see that “counter” can be optimized fully and the compiler can replace the variable “counter” with 54 wherever counter is used to generate faster code. However, “var2” is not set to a compile time constant value, so for “var2” we need to wait until runtime to bind its value. While runtime constants are still faster than normal variables, their speed is not as good. To prove these are both constants, any write to them will produce errors:

```
#include <iostream>
using namespace std;

int main() {
  const int counter = 54;
  counter++;

  int var1 = 3;
  const int var2 = var1;
  var2++;
}
```

```
$ g++ constants.cpp
constants.cpp: In function 'int main()':
constants.cpp:6:3: error: increment of read-only variable 'counter'
    6 |   counter++;
      |   ^~~~~~~
constants.cpp:10:3: error: increment of read-only variable 'var2'
   10 |   var2++;
      |   ^~~~
```

C# also has both compile time and runtime constants, although it makes this distinction clearer. It uses the “const” keyword for a compile time constant and the “readonly” keyword for a runtime constant:

```
public class FrameSize {
  public int size;

  public FrameSize(int size) {
    this.size = size;
  }
}

public class PictureFrame {
  public readonly FrameSize frameSize;
  public const bool hasGlass = true;

  public PictureFrame(int size) {
    frameSize = new FrameSize(size);
  }
}

public class Program {
  public static void Main(string[] args) {
    PictureFrame frame = new PictureFrame(3);
    System.Console.WriteLine(frame.frameSize.size);
  }
}
```

The value of the field “hasGlass” can be known during compilation, so we use “const” on it to make it a compile time constant. The value of the field “frameSize” can only be known at runtime, so we use “readonly” on it to make it a runtime constant. This is true even if we already know the value, but that value has to be dynamically created using the “new” keyword:

```
public class FrameSize {
  public int size;

  public FrameSize(int size) {
    this.size = size;
  }
}

public class PictureFrame {
  public readonly FrameSize frameSize = new FrameSize(3);
  public const bool hasGlass = true;

  public PictureFrame() {
  }
}

public class Program {
  public static void Main(string[] args) {
    PictureFrame frame = new PictureFrame();
    System.Console.WriteLine(frame.frameSize.size);
  }
```

```
}
```

Here the value of “frameSize” still must be created at runtime (i.e. we can’t instantiate objects during compilation). To prove the “readonly” keyword is needed, what happens if we use “const” instead:

```
public class FrameSize {
  public int size;

  public FrameSize(int size) {
    this.size = size;
  }
}

public class PictureFrame {
  public const FrameSize frameSize = new FrameSize(3);
  public const bool hasGlass = true;

  public PictureFrame() {
  }
}

public class Program {
  public static void Main(string[] args) {
    PictureFrame frame = new PictureFrame();
    System.Console.WriteLine(frame.frameSize.size);
  }
}
```

```
$ csc constants.cs
Microsoft (R) Visual C# Compiler version 3.11.0-4.22108.8 (d9bef045)
Copyright (C) Microsoft Corporation. All rights reserved.

constants.cs(10,38): error CS0133: The expression being assigned to
'PictureFrame.frameSize' must be constant
```

As you can see, “const” cannot be used if the value of the constant variable is dynamically created. To make a variable “const”, the value must be able to be computed/evaluated fully during compilation.

Ok, enough talk of constant variables, what about methods? What does putting the word “final” on methods do? To understand this, we need to revisit polymorphism.

Recall that polymorphism allows an object to behave differently depending on its specific data type. For example:

```
class Computer {
  public void turnOn() {
    System.out.println("Turned on successfully");
  }
}

class HP extends Computer {
  @Override
  public void turnOn() {
    System.out.println("Showing HP splash screen");
  }
}

class Dell extends Computer {
  @Override
  public void turnOn() {
    System.out.println("Catching fire and dying");
  }
}

public class PolymorphismTest {
  public static void main(String[] args) {
    Computer c1 = new Computer();
    Computer c2 = new HP();
    Computer c3 = new Dell();

    c1.turnOn();
    c2.turnOn();
    c3.turnOn();
  }
}
```

As you can see, all three variables are a “Computer” object, yet they have different specific types and so the runtime determines specifically which “turnOn” method is invoked in each case. I will cover this in more detail in a later lecture (including defining a term for a variable’s “specific type”), but for now this is all you need to know.

What if you didn’t want to allow a method from your class to be overridden? By default, any method you create in a Java class can be overridden by a subclass, unless you use the “final” keyword:

```
class Computer {
  public final void turnOn() {
    System.out.println("Turned on successfully");
  }
}

class HP extends Computer {
  @Override
  public void turnOn() {
    System.out.println("Showing HP splash screen");
  }
}

class Dell extends Computer {
  @Override
  public void turnOn() {
    System.out.println("Catching fire and dying");
  }
}

public class PolymorphismTest {
  public static void main(String[] args) {
    Computer c1 = new Computer();
    Computer c2 = new HP();
    Computer c3 = new Dell();

    c1.turnOn();
    c2.turnOn();
    c3.turnOn();
  }
}
```

```
$ javac PolymorphismTest.java
PolymorphismTest.java:9: error: turnOn() in HP cannot override turnOn() in
Computer
  public void turnOn() {
              ^
  overridden method is final
PolymorphismTest.java:16: error: turnOn() in Dell cannot override turnOn()
in Computer
  public void turnOn() {
              ^
  overridden method is final
2 errors
```

Applying the “final” keyword on a method means that the method cannot be overridden by a subclass and javac will complain if you attempt to. This doesn’t mean you can’t inherit from the Computer class, just that you can’t override its final method “turnOn”. If you want to prevent subclassing all together, use the final keyword on the class itself:

```
final class Computer {
  public void turnOn() {
    System.out.println("Turned on successfully");
  }
}

class HP extends Computer {
  @Override
  public void turnOn() {
    System.out.println("Showing HP splash screen");
  }
}
class Dell extends Computer {
  @Override
  public void turnOn() {
    System.out.println("Catching fire and dying");
  }
}

public class PolymorphismTest {
  public static void main(String[] args) {
    Computer c1 = new Computer();
    Computer c2 = new HP();
    Computer c3 = new Dell();

    c1.turnOn();
    c2.turnOn();
    c3.turnOn();
  }
}
```

```
$ javac PolymorphismTest.java
PolymorphismTest.java:7: error: cannot inherit from final Computer
class HP extends Computer {
                 ^
PolymorphismTest.java:14: error: cannot inherit from final Computer
class Dell extends Computer {
                   ^
2 errors
```

As stated previously, methods in Java are allowed to be overridden by default. However, in languages like C++ and C#, this is exactly the opposite. In those languages, methods are not allowed to be overridden unless you use the keyword “virtual”. Here is an example in C#:

```
public class Computer {
  public void TurnOn() {
    System.Console.WriteLine("Turned on successfully");
  }
}

public class HP : Computer {
  public override void TurnOn() {
    System.Console.WriteLine("Showing HP splash screen");
  }
}

public class PolymorphismTest {
  public static void Main(string[] args) {
    Computer c1 = new Computer();
    Computer c2 = new HP();
    c1.TurnOn();
    c2.TurnOn();
  }
}
```

```
$ csc finalmethods.cs
Microsoft (R) Visual C# Compiler version 3.11.0-4.22108.8 (d9bef045)
Copyright (C) Microsoft Corporation. All rights reserved.

finalmethods.cs(8,24): error CS0506: 'HP.TurnOn()': cannot override
inherited member 'Computer.TurnOn()' because it is not marked virtual,
abstract, or override
```

C# is complaining here because I'm trying to override a method that, by default, is not able to be overridden. I'll have to add the "virtual" keyword to make this work:

```
public class Computer {
  public virtual void TurnOn() {
    System.Console.WriteLine("Turned on succesfully");
  }
}

public class HP : Computer {
  public override void TurnOn() {
    System.Console.WriteLine("Showing HP splash screen");
  }
}

public class PolymorphismTest {
  public static void Main(string[] args) {
    Computer c1 = new Computer();
    Computer c2 = new HP();
    c1.TurnOn();
    c2.TurnOn();
  }
```

```
}
```

So, in summary, Java methods are virtual by default and adding the “final” keyword takes away the ability to override them. C++ and C# methods are final by default and adding the “virtual” keyword gives back the ability to override them.

To summarize what the final keyword can do (with respect to Java), here is a table:

Apply “final” keyword to:	**Has the effect of:**
local variable or field	becomes a compile time or runtime constant and value can’t change once initialized
method	disallows overriding in subclasses
class	disallows inheriting from this class (can’t create subclasses)