CS 360
Programming Languages
Introduction to Java

The plan

- Racket will return!
  - Final project will be writing a Racket interpreter *in Java*.

- Lecture will not discuss every single feature of Java.
  - You may need to do some digging on your own.
  - Lots of help online (Google is your friend).

Java Resources

- Java tutorial
  - http://docs.oracle.com/javase/tutorial/java
- Java documentation
  - http://docs.oracle.com/javase/8/docs/api
- And if you're confused about anything, Google will find it.
  - There's so much Java stuff on the web because most undergraduate curriculums now teach Java as their first or second language.

Logistics

- We will use Java version 8.
  - Though probably most of the code I will show is compatible back to Java 6 and 7.
  - Java 9 was just released about six weeks ago.
- Many powerful IDEs out there.
  - I will be using an IDE called NetBeans, which is free.
  - Installation instructions will be on the class webpage.

Next Assignments

- Overlapping time frames for the last assignments.
- Project 4 – out today, still in Racket
  - Out today
  - Due Tue Nov 14
- Project 5 – Java warmup assignment
  - Out Thu Nov 9
  - Due Tue Nov 21 [day before Thanksgiving break].
- Project 6 – Java project involving threads and concurrency
  - Out Tue Nov 14
  - Due Tue Nov 28
- Project 7 – Racket interpreter in Java.
  - Out Tue Nov 28
  - Due during final exams (probably Tue Dec 12).

History of Java

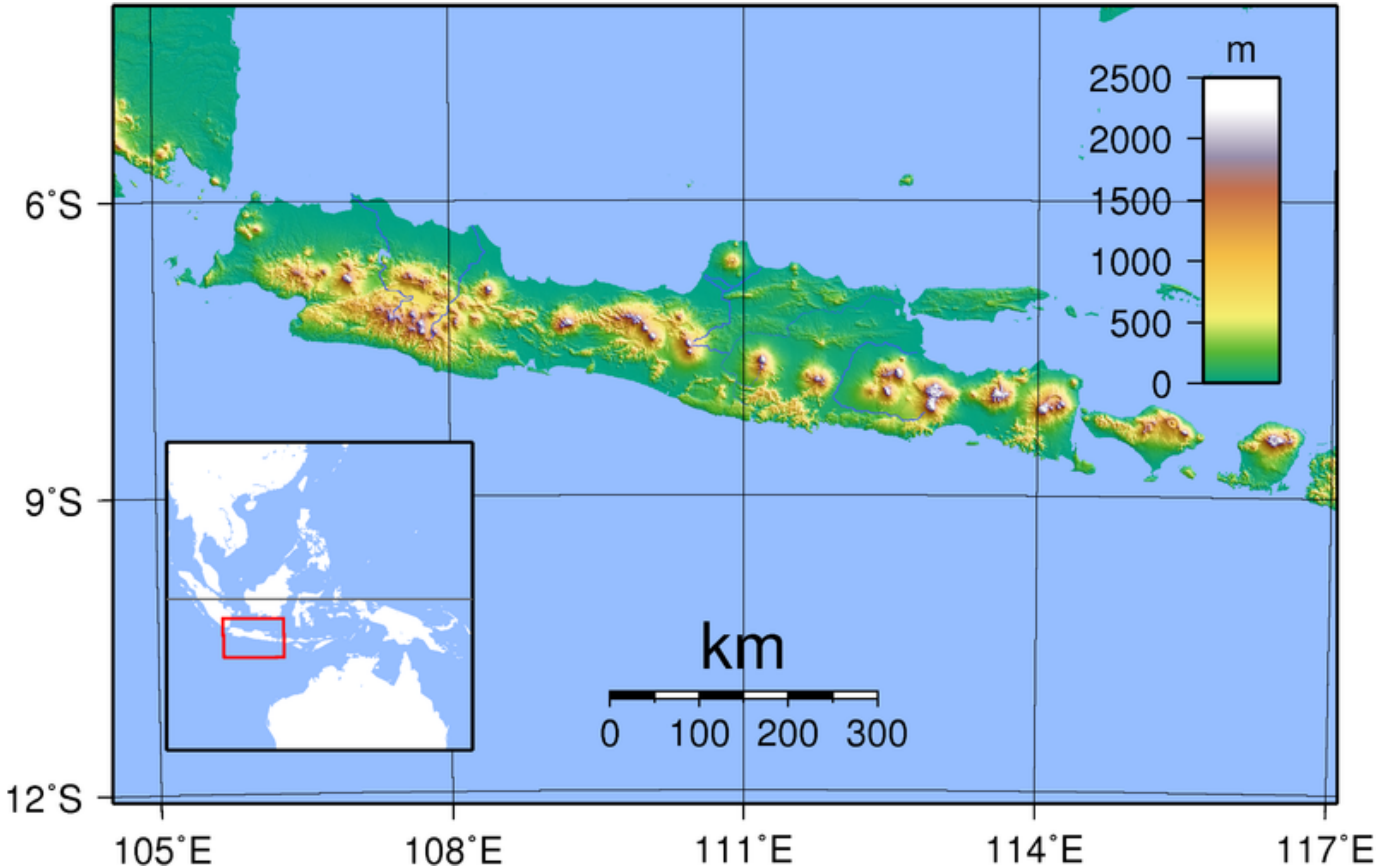
History of Java

- Java was first used in the 15th century, in Yemen, and quickly spread to Egypt and North Africa.

The Real History of Java

The Real History of Java

- Java is millions of years old and 135 million people see Java every day.


m
2500
2000
1500
1000
500
0
6°S
9°S
12°S
105°E
108°E
111°E
114°E
117°E
km
0
100
200
300

The Real, Real History of Java

- The Java project was initiated at Sun Microsystems in 1991.
  - Supposedly named after the large quantities of coffee the language designers drank.

- Originally was designed to be embedded in consumer electronic devices, like cable TV set-top boxes, but it was too advanced for the cable television industry at the time.

- Language evolved into a general-purpose programming language.

The Real, Real History of Java

- Java was designed to use a syntax similar to C and C++.
  - Lots will be familiar.

- Java is (almost completely) object oriented.
  - All data types are classes, except for the primitives like int, long, float, double, char, boolean.
  - All code is written inside some class.
    - All functions are methods (no free-floating functions).
  - Single inheritance only (C++ allows multiple).

- Statically typed (like C++,).

- Has *generics* (similar to C++ templates).

The Real, Real History of Java

- Same basic programming properties as C++.
  - Must declare variables before use, say what type they are.
  - If/else, for, while, do-while, switch work just like C++.

- No pointers!
  - Java uses a similar idea called references, which are "safer" than pointers.

- All objects stored on the heap (using "new").

- Garbage collection
  - No explicit allocation/deallocation of memory. (no malloc/free)

Defining a class

- Take a look at the Rational class.

- Create primitive variables just like in C++:
  - int x = 4;
  - float f = 3.02;
  - boolean b = true; // note lowercase

- Strings are objects, but Java lets you create them like a primitive:
  - String s = "a wonderful string";

- All other objects are created using new:
  - ClassName var = new ClassName(args);
  - Constructor automatically chosen based on data types of arguments.

- Variables declared in a class are called ***fields*** or ***instance variables.*** (like C++)

- Instances of a class have one copy of their fields or instance variables.

- Contrast with ***class variables*** or ***static variables***: one copy of the variable that is shared among all instances of the class.
  - Declared with **`static`** keyword.

- Functions declared in a class known as *methods*.

- ***Instance methods*** can access instance variables, and are called using C++-like syntax:
  - `ClassName var = new ClassName();`
  - `var.name_of_method(arg1, arg2, ...);`

- ***Class methods*** or ***static methods*** are called on the name of the class itself, not an instance of the class.
  - `ClassName.name_of_instance_method(args);`
  - example: `Integer.toString(int)`, `Math.pow(x, y)`

Class/Method/Variable Visibility

- **`public`**: available everywhere
- **`protected`**: only available within the class and subclasses
- **`private`**: only available within the class

- Similar to C++:
  - Have a number of private instance variables that maintain the "state" of the class.
  - Have a number of public methods that are part of the class's interface.
  - Also common to have private "helper" methods.

- Java traditionally uses CamelCase rather than separating_with_underscores.
- variables and methods start with a lowercase letter.
- Class names start with an uppercase letter.
- "this" works just like in C++.
- All objects by default inherit from the "Object" base class.

Getting a program started

- Each class must go in its own file, which must be named ClassName.java.
- Any class can have a public static main() method, which is where the execution starts.

Packages

- Java's standard library (all the functions that the language comes with) are organized into packages
  - A hierarchical organization system.

- In Java you "import" classes from packages, whereas in C++ you "#include" files.

Collections

- Built in classes for
  - Lists (ArrayList, LinkedList, …)
  - Sets (HashSet, …)
  - Maps (what Java calls hash tables) (HashMap)

- All of these are parameterized with generics.
  - List<Integer> intlist = new List<Integer>();
  - intlist.add(17);
  - System.out.println(intlist); // prints [17]

Today's plan

- Introduce OOP concepts from the ground up using Java.
  - Rehash of 142-ish things but at a deeper level of understanding.

- Talk about ***why/when you should or shouldn't*** do certain OOP things.

- Lots of things will be familiar from C++.

- Some things will be different.

```
public class Point
{
  private int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public double distFromOrigin() {
    return Math.sqrt(x * x + y * y)
  }
}
```

Subclassing

- A class definition has a *superclass* (**`Object`** if not specified)

```
class ColorPoint extends Point { … }
```

- The superclass affects the class definition:
  - Class *inherits* all field declarations from superclass
  - Class *inherits* all private method definitions from superclass
    - Code within the subclass cannot directly access any private fields or methods.
  - But class can *override* method definitions as desired

```java
public class ColorPoint extends Point
{
  private Color color;
  public ColorPoint(int x, int y, Color c) {
    super(x, y); // call the superclass constructor
    this.color = c;
  }
  public Color getColor() { return color; }
  public void setColor(Color c) { this.color = c; }
}
```

An object has a class

- Using instanceof can indicate bad OO style.
  - If you're using it to do something different for different objects types, you probably meant to write a method and have subclasses override the method.
- instanceof is an example of using reflection
  - Reflection is the ability for a computer program to be able to examine its structure and behavior at run-time.

```
Point p = new Point(0, 0);
ColorPoint cp = new ColorPoint(0, 0, Color.red)

/* instanceof is a keyword that returns true
   if a variable is an instance of a class. */

p instanceof Point       // true
cp instanceof ColorPoint // true
cp instanceof Point      // true
```

Why subclass?

- Instead of creating **`ColorPoint`**, could add methods to **`Point`**
  - That could mess up other users and subclassers of **`Point`**

```
public class Point {
  private int x, y;
  private Color color;
  …

  public Point(x, y) {
    // what does color get set to?
  }
}
```

Why subclass?

- Instead of subclassing **`Point`**, could copy/paste the methods
  - Means the same thing *if* you don't use **`instanceof`**, but of course code reuse is nice

```
public class ColorPoint {
  private int x, y;
  private Color color;
  …
}

ColorPoint cp = new ColorPoint( whatevs )
if (cp instanceof Point) {
  // do pointy things
}
```

Why subclass?

- Instead of subclassing **`Point`**, could use a **`Point`** instance variable inside of ColorPoint.
  - Define methods to send same message to the **`Point`**
  - This is called object composition; expresses a "has a" relationship.
  - But for **`ColorPoint`**, subclassing makes sense: less work and can use a **`ColorPoint`** wherever code expects a **`Point`**

```
public class ColorPoint {
  private Point point;
  private Color color;
  public setX(int x) { point.setX(x); }
  …
}
```

Is-a vs has-a

- OO beginners tend to overuse inheritance (the is-a relationship).

- OO inheritance is notoriously tricky to get right sometimes (e.g., writing methods that test for equality)
  - boolean equals(Point a, Point b)
  - What if a & b can be Points or ColorPoints?

- Many real-world relationships can be expressed using is-a or has-a, even if the most natural way seems to be is-a.
  - ColorPoint could be written using object composition.

Circle and ellipse problem

- What should the relationship be between a Circle class and an Ellipse class?

Circle and ellipse problem

- Circles are specific types of ellipses, so a Circle **is-a** Ellipse.

```
public class Ellipse {
  private int radiusX, int radiusY;
  public void setRadiusX(int rx) { radiusX = rx; }
  public void setRadiusX(int rx) { radiusY = ry; }
  public int getRadiusX() { return radiusX; }
  public int getRadiusY() { return radiusY; }
}
public class Circle extends Ellipse {
  …
}
```

Circle and ellipse problem

- Circles are specific types of ellipses, so a Circle **is-a** Ellipse.
- But now Circle has a setRadiusX() method.
- Furthermore, what would that method's implementation look like?

Circle and ellipse problem

- Different solution: make Ellipse a subclass of Circle.
  - "An Ellipse is a Circle with an extra radius field."

```
public class Circle {
  private int radius;
  public void setRadius(int r) { radius = r; }
  public int getRadius() { return radius; }
}
public class Ellipse extends Circle {
  private int radiusY;
  // assume existing radius is for X dimension.
}
```

Circle and ellipse problem

- Different solution: make Ellipse a subclass of Circle.
  - "An Ellipse is a Circle with an extra radius field."

- Just as many problems here:

- What does it mean when an Ellipse calls Circle's setRadius or getRadius method (which radius?)

One solution: Immutability

- Let Circle inherit from Ellipse and eliminate mutator methods.

```
public class Ellipse {
  private int radiusX, int radiusY;
  public int getRadiusX() { return radiusX; }
  public int getRadiusY() { return radiusY; }
}

public class Circle extends Ellipse { … }
```

- Circle still has two radius accessor methods.
- As long as Circle's constructor forces radiusX = radiusY, there's no way to violate that constraint later.

Other solutions

- Let Circle and Ellipse inherit from some common superclass (rather than one from the other).

- Let setRadiusX() return success or failure.

- Drop inheritance entirely.

- Drop Circle; let users (manually) handle circles as instances of Ellipse.

What inheritance really is for

- Inheritance gets you into trouble when it seems like the relationship is "is-a," but it actually is "is-a-restricted-version-of."
  - Circle and Ellipse
  - Person and Toddler
    - Certainly a Toddler is a Person.
    - But what if a Person has a method called walk(int distance).
    - Toddlers can't walk!

- Inheritance should be used to add extra detail to a superclass (e.g., a Monkey is an Animal), not to restrict functionality.
  - ColorPoint is (probably) fine to inherit from Point

Try this one out

- I want to declare a class ThreeDPoint.

- Should this inherit from Point?
  - What are the pros and cons?

Something different: Method overriding

- In OOP, a subclass may override a method from a superclass.
- Just re-define the method in the subclass.

In C++, what does this do?

```
class Base {
  public: int f() { return 1; } };
class Derived: public Base {
  public: int f() { return 2; } };

int main() {
  Base b;
  Derived d;
  cout << b.f() << endl;
  cout << d.f() << endl;
  b = d;
  cout << b.f() << endl;
  Base *b2 = &d;
  cout << b2->f() << endl;
}
```

```
Base *b2 = &d;
  cout << b2->f() << endl;
```

- With a pointer to an object, a call to a method of that object calls the version of the method *specified by the type of the pointer*, not the type of the object being pointed to.

- Can be changed with the C++ keyword **`virtual`**.

- With a pointer to an object, a call to a virtual method of that object calls the version of the method *specified by the type of the object being pointed to*.

In C++, what does this do?

```
class Base {
  public: virtual int f() { return 1; } };
class Derived: public Base {
  public: int f() { return 2; } };

int main() {
  Base b;
  Derived d;
  cout << b.f() << endl;
  cout << d.f() << endl;
  b = d;
  cout << b.f() << endl;
  Base *b2 = &d;
  cout << b2->f() << endl;
}
```

- The key idea here is called ***dynamic dispatch:***
  - Selecting which implementation of a polymorphic operation to call at ***run-time***, rather than ***compile-time.***
- This is the opposite of what we've learned about lexical (static) scope:
  - In lexical scope, we always know at compile-time what variables will be referred to and what functions will be called.
- With OOP, it is possible for a variable to refer to an object whose type is uncertain at compile time.

```
Base b;
Derived d;
Base *b2 = nullptr;
if (rand() > 0.5))
  b2 = &b;
else
  b2 = &d;
b2->f();
```

Java virtual methods

- In Java, all methods are virtual.
  - This behavior cannot be changed.
  - If a subclass needs to call a superclass's version of an overridden method from a subclass, there is the **`super`** keyword:

```
public class Base {
  public int f() { return 1; } }
public class Derived extends Base {
  public int f() { return 2 + super.f(); } }
```

Java virtual methods

```
public class ThreeDPoint extends Point
{
  private int z;

  // override distFromOrigin in Point
  public double distFromOrigin() {
    return Math.sqrt(
     getX()*getX() + getY()*getY() + z*z;
  }
}
```

Java I/O

- Main way of outputting to the screen:

- **`System.out.println(x);`**
  - takes one argument of any type
  - if x is an object, its **`toString()`** method will be automatically called to convert it to a String.
  - also **`System.err.println(x)`**;

  - System.out is an OutputStream object (similar to **`cout`** in C++)

Java I/O

- There are about 50 bazillion ways to do input in Java.
- Easiest way:
  - **`import java.util.*;`**
  - **`Scanner scanner = new Scanner(System.in)`**
    - System.in is an InputStream object (similar to **`cin`** in C++)
  - Now call any of the following:
  - **`scanner.nextInt()`** [or nextLong(), nextFloat(), etc]
    - all of these stop at the first whitespace found
  - **`scanner.nextLine()`**
    - reads a whole line, returns a String

Try this

- Make a program that reads in integers from the keyboard until you enter -1.

Collections

- Java has many collection classes.
  - ArrayList, HashSet, HashMap most common.
  - Very few cases where you need "real" arrays; using ArrayList is much more common.

- Syntax is similar to C++ templates
  - e.g., C++'s vector, set, and map

- Gotcha: Only objects can be stored in Java's collection classes.
  - No ints, floats, booleans, doubles, etc in ArrayLists!
  - Java has "wrapper" classes Integer, Float, Boolean, Double that you use instead, and Java does the conversion for you.

ArrayList (example for ints)

- Creation
  - `List<Integer> list = new ArrayList<Integer>();`
- Put stuff in
  - `list.add(x);    // adds x to end by default`
  - `list.add(i, x); // inserts x at list[i]`
  - `list.set(i, x); // changes list[i] to x`
- Get stuff out
  - `list.get(i); // returns list[i]`
- Other stuff
  - `list.size()`, `list.contains(x)`, `list.indexOf(x)`, `list.remove(i)`,

Enhanced for loop

```
for (int i = 0; i < list.size(); i++) {
  System.out.println(list.get(i));
}

for (int x : list) {
  System.out.println(x);
}
```

Try this

- Make a program that reads in integers from the keyboard until you enter -1.
- Add all the integers (as they're entered) to an ArrayList.
- Print out all the integers. Try this two ways:
  - System.out.println(list);
  - With the enhanced for loop.

Try this

- Make a program that reads in integers from the keyboard until you enter -1.
- Add a static method fib(n) that computes the n'th Fibonacci number. Write this the standard (slow, recursive) way.
- Print out the Fibonacci value of each number as they're entered.
  - What is the max Fibonacci # you can compute before you get an error?

HashMaps

- Java's has a few hashtable classes.
- Most common is HashMap.

- The Java language was constructed with hashtables in mind.
- The Object class has a hashCode() method.
  - Because all objects inherit (directly or indirectly) from Object, all classes have a hashCode() method!
- If you ever make a class that you want to use as the key of a hashtable, you should override the hashCode() and equals() methods.
  - Don't worry about this at the moment.

HashMap (example for String map to int)

- Creation
  - **Map<String, Integer> map
    = new HashMap<String, Integer>();**
- Put stuff in
  - **map.put(s, i); // associates key s with value i**
- Get stuff out
  - **map.get(s); // returns whatever value s is associated with**
- Other stuff
  - **map.size(), map.containsKey(s), map.keySet(), map.remove(s)**

Enhanced for loop

You can use the enhanced for loop to iterate through a map:

```
for (String key : map.keySet()) {
  int value = map.get(key);
  // do something with key and/or value
}
```

Try this: memoized Fibonacci in Java

- Add a HashMap<Integer, Integer> as a static field to your class.
  - This will store the cached Fibonacci values.
- Alter your Fibonacci method so it does the following:
  - For fib(n):
  - if n = 0 or n = 1, return n
  - Check if n is a key in the hashtable.
    - If it is, get the corresponding value and return it.
    - If it's not, then
      - compute v = fib(n-1) + fib(n-2)
      - put the mapping from n to v in the hashtable
      - return v

HashSets

- A Set (ADT) is an *unordered* collection of items.
  - A List is an *ordered* collection of items.
- Java has a HashSet class that implements this ADT.
- Similar to C++'s std::set class.

HashSet (example for ints)

- Creation
  - `HashSet<Integer> set = new HashSet<Integer>();`
- Put stuff in
  - `set.add(x);      // adds x to the set`
- Test if something is in the set
  - `set.contains(x);    // returns true or false`
- Remove something from the set
  - `set.remove(x);`
- Other stuff
  - `set.size(), set.isEmpty(), set.clear()`