In theory, you already know Java...

- Classes and objects
- Static vs instance fields and methods
- Primitive vs reference types
- Private vs public vs package
- Constructors
- Method signatures
- Local variables
- Arrays
- Subtypes and Inheritance, Shadowing

... but even so

- Even standard Java features have some subtle aspects relating to object orientation and the way the type system works
- Let’s touch on a few of these today
- We picked topics that will get you thinking about Java the way that we think about it!

Java is object oriented

- In most prior languages, code was executed line by line and accessed variables or record

- In Java, we think of the data as being organized into objects that come with their own methods, which are used to access them
  - This shift in perspective is critical
  - When coding in Java one is always thinking about “which object is running this code?”

Dynamic and Static

- Some kinds of information is “static”
  - There can only be one instance
  - Like a “global variable” in C or C++ (or assembler)

- Object-oriented information is more “dynamic”
  - Each object has its own private copy
  - When we create a new object, we make new copies of the variables it uses to keep its state

- In Java this distinction becomes very important

Names

- The role of a name is to tell us
  - Which class is being referenced, although sometimes this is clear from the context
  - Which object is being referenced, unless we’re talking about a static method or a static variable

- Example
  - `System.out.println(a.serialNumber)`
    - `out` is a static field in class `System`
    - The value of `System.out` is an instance of a class that has an instance method `println(int)`

- If an object must refer to itself, use `this`
  - `this.i = i;

The main Method

```
public static void main(String[] args) {
...
}
```

- Can be called from anywhere
- Associated with the class, don’t need an instance (an object) to invoke it
- No return value
- Method must be named `main`
- Parameters passed to program on command line or, in Eclipse, can be defined in the “Run” configuration dialog box (which is the same as the “Debug” one…)
Static methods and variables

Class Foo

```java
class Foo {
    static int x;
    static void bar(int i) {
        ...
    }
    void something() {
        ...
    }
}
```

Foo instance `a`

```java
Foo a = new Foo();
```

Dynamic method `something()`

```java
Dynamic method
```

Class Foo

```
static method
```

Dynamic variable `y`

```
dynamic variable
```

Avoiding trouble

- Use of static methods is discouraged
- Keep in mind that "main" is a static method
  - Hence anything main calls needs to have an associated object instance, or itself be static

```java
class Thing {
    static int s_val;         // One for the whole class
    int o_val;                // Each object will have its own personal copy
    static void s_method()  // Anyone can call this
    {
        s_val++;            // Legal: increments the shared variable s_val
        o_val = s_val;      // Illegal: Which version of s_val do we mean?
        o_method(s_val);   // Illegal: o_method needs an object reference
    }
    void o_method() {
        s_val++;        // Legal
        this.s_val++;   // Illegal: s_val belongs to the class, not object
        o_val = s_val;  // Legal: same as this.s_val = s_val
        s_method();    // Legal: calls the class method s_method()
        o_method();    // Legal: same as this.o_method();
    }
    } // end class Thing
```

Constructors

- Called to create new instances of a class
- Default constructor initializes all fields of the class to default values (0 or null)

```java
class Thing {
    int val;
    Thing(int val) {
        this.val = val;
    }
    Thing() {
        this(3);
    }
}
```

```java
Thing one = new Thing(1);
Thing two = new Thing(2);
Thing three = new Thing(3);
```

What about non-class variables?

- Those are not automatically initialized, you need to do it yourself!
- Can cause confusion

```java
class Thing {
    int val;
    Thing(int val) {
        this.val = val + undef;
    }
    Thing() {
        this(3);
    }
}
```

```java
this.val was automatically initialized to zero, but undef has no defined value!
Yet the declaration looks very similar!
In what way did it differ?
```

Finalizers

- Like constructors but called when the object is deallocated
- Might not happen when you expected
  - Garbage collector decides when to actually deallocate an object
  - So objects can linger even when you no longer have a reference to them!
  - For this reason, we tend not to use finalizers — they add an undesired form of unpredictability
Static Initializers

- Run once when class is loaded
- Used to initialize static objects

```java
class StaticInit {
    static Set<String> courses = new HashSet<String>();
    static {
        courses.add("CS 2110");
        courses.add("CS 6110");
    }
    public static void main(String[] args) {
        //...
    }
}
```

Names

- Refer to my static and instance fields & methods by (unqualified) name:
  - serialNumber
  - Widget.nextSerialNumber
- Refer to static fields & methods in another class using name of the class
  - Widget.nextSerialNumber
- Refer to instance fields & methods of another object using name of the object
  - a.serialNumber

Overloading of Methods

- A class can have several methods of the same name
  - But all methods must have different signatures
  - The signature of a method is its name plus the types of its parameters
- Example: String.valueOf(...) in Java API
  - There are 9 of them:
    - valueOf(boolean);
    - valueOf(int);
    - valueOf(long);
  - ...
- Parameter types are part of the method’s signature

Example: Overloading “compareTo”

- Many classes extend an object that supports an interface called “comparable”. If you do this you can override these methods:
  - equals(): a.equals(b), returns true/false
  - compareTo(): a.compareTo(b): returns -/0/+ 
  - hashCode(): a.hashCode; usually you return data.hashCode() for some data object in a that represents a’s “value” (perhaps a string or a number)
- Overriding all three methods allows Java utilities that sort arrays to operate on your class
- But one warning: if you override these methods you must override all of them

Primitive vs Reference Types

- Primitive types
  - int, short, long, float, byte, char, boolean, double
  - Efficient
  - 1 or 2 words
  - Not an Object—unboxed
- Reference types
  - Objects and arrays
  - String, int[], HashSet
  - Usually require more memory
  - Can have special value null
  - Can compare null with ==, !=
  - Generates NullPointerException if you try to dereference null
Comparing Reference Types

- Comparing objects (or copying them) isn't easy!
  - You need to copy them element by element
  - Compare objects using the "equals" method, which implements "deep equality"

- Example: suppose we have
  - Person A = new Person("Fred"), B = new Person("Fred");
  - What will A == B return?
  - Need to use A.equals(B) False! A and B are different Persons even though their values are the same. (Beware that this ought to be the same behaviour for Strings, but isn’t since they’re immutable!)

== with primitive types

- Puzzle: why do integer comparisons work?
  - Integer I = 7;
  - (I == 7)? True, but not obvious why!
  - (I == new Integer(7)) False

... the first comparison only works because Java auto-unboxes I to compare it with int 7.
- If it had autoboxed the 7, the comparison would have failed! Lucky Java gets this right...

Arrays

- Arrays are reference types
  - Array elements can be reference types or primitive types
    - E.g., int[] or String[] or Rhubarb[]
  - a is an array, a.length is its length
  - Its elements are
    - a[0], a[1], ..., a[a.length-1]
  - The length is fixed when the array is first allocated using « new »

Accessing Array Elements Sequentially

```java
public class CommandLineArgs {
    public static void main(String[] args) {
        System.out.println(args.length);
        // old-style
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
        // new style
        for (String s : args) {
            System.out.println(s);
        }
    }
}
```

Class Hierarchy

- Every class (except Object) has a unique immediate superclass, called its parent
- Object is the ultimate superclass
Inheritance

- A subclass inherits the methods of its superclass
- Example: methods of the Object superclass:
  - equals(), as in A.equals(B)
  - toString(), as in A.toString()
  - ... others we'll learn about later in the course
- ... every object thus supports toString()!

Overriding

- A method in a subclass overrides a method in superclass if:
  - both methods have the same name,
  - both methods have the same signature (number and type of parameters and return type), and
  - both are static methods or both are instance methods
- Methods are dispatched according to the runtime type of the actual, underlying object (cf example coming up!!!)

Accessing Overridden Methods

- Suppose a class Child overrides a method m in its parent
- Methods in Child can invoke the overridden method in the parent as
  - super.m()
- In particular, can invoke the overridden method in the overriding method! This can be very useful.
- Caveat: cannot compose super more than once as in
  - super.super.m() ---- not kosher!

Unexpected Consequences

- An overriding method cannot have more restricted access than the method it overrides

```java
class A {
    public int samename() {...}
}
class B extends A {
    private int samename() {...} // illegal !
}
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                                                      // Shadows variable i in class A.
    int f() { return -i; }                                        // Overrides method f in class A.
}
```

A foo = new B(); // upcasting
foo.samename();  // would invoke private method in class B at runtime, were it to be legal

... a nasty example

```java
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                                                      // Shadows variable i in class A.
    int f() { return -i; }                                        // Overrides method f in class A.
}
```

public class Override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);                       // Refers to B.i; prints 2.
        System.out.println(b.f());                     // Refers to B.f(); prints -2.
        A a = (A) b;                                            // Cast b to an instance of class A.
        System.out.println(a.i);                        // Now refers to A.i; prints 1;
        System.out.println(a.f());                     // Still refers to B.f(); prints -2;
    }
}

Shadowing

- Like overriding, but for fields instead of methods
  - Superclass variable v of some type
  - Subclass: variable v perhaps of some other type
  - Method in subclass can access shadowed variable using super.v
- Variable references are resolved using static binding (i.e., at compile-time), not dynamic binding (i.e., not at runtime)
- Variable reference r.v uses the static (declared) type of the variable
- r, not the runtime type of the object referred to by r
- Shadowing variables is bad medicine and should be avoided
... back to our earlier example

```java
class A {
    int i = 1;
    int f() { return i; }
}

class B extends A {
    int i = 2;  // Shadows variable i in class A.
    int f() { return -i; }  // Overrides method f in class A.
}

public class override_test {
    public static void main(String args[]) {
        B b = new B();  // Refers to B.i; prints 2.
        System.out.println(b.i);  // Refers to B.i; prints -2.
        System.out.println(b.f());  // Refers to B.f(); prints -2.
        A a = (A) b;  // Cast b to an instance of class A.
        System.out.println(a.i);  // Now refers to A.i; prints 1;
        System.out.println(a.f());  // Still refers to B.f(); prints -2;
    }
}
```

The "declared" or "static" type of "a" is "A"!

### Array vs ArrayList vs HashMap

#### Array
- Storage is allocated when array created; cannot change
- Extremely fast lookups

#### ArrayList (in java.util)
- An "extensible" array
- Can append or insert elements, access i'th element, reset to 0 length
- Lookup is slower than an array

#### HashMap (in java.util)
- Save data indexed by keys
- Can lookup data by its key
- Can get an iteration of the keys or values
- Storage allocated as needed but works best if you can anticipate need and tell it at creation time.

### HashMap Example

- Create a HashMap of numbers, using the names of the numbers as keys:
  ```java
  Map<String, Integer> numbers = new HashMap<String, Integer>();
  numbers.put("one", new Integer(1));
  numbers.put("two", new Integer(2));
  numbers.put("three", new Integer(3));
  ```
- To retrieve a number:
  ```java
  Integer n = numbers.get("two");
  ```
- Returns null if the HashMap doesn't contain key
- Can use numbers.containsKey(key) to check this

### Generics and Autoboxing

- Old (pre-Java 5)
  ```java
  Map numbers = new HashMap();
  numbers.put("one", new Integer(1));
  Integer s = (Integer) numbers.get("one");
  ```
- New (generics)
  ```java
  Map<String, Integer> numbers = new HashMap<String, Integer>();
  numbers.put("one", new Integer(1));
  Integer s = numbers.get("one");
  ```
- New (generics + autoboxing)
  ```java
  Map<String, Integer> numbers = new HashMap<String, Integer>();
  numbers.put("one", 1);
  int s = numbers.get("one");
  ```

### Experimentation

- All of this adds up to some pretty confusing stuff you'll need to learn!
- Don't be afraid to experiment by writing little code fragments and seeing if they compile and what they do.
- But don't write random code hoping that it might work by some miracle.
- Examples in the Sun online JDK manual can be really helpful
  ```java
  Cut and paste from Sun JDK manual is not considered to be a violation of academic integrity.
  So go for it!
  ```

### Debugging

- Do not just make random changes, hoping something will work. This never works.
- Think about what could cause the observed behavior
- Isolate the bug. Focus on the first thing that goes wrong.
- An IDE helps by providing a Debugging Mode
  ```java
  Can set breakpoints, step through the program while watching chosen variables
  When program pauses at breakpoint, or dies, can look at values of variables it was using
  ```