Objects and Classes

CS211
Fall 2000

Much of Java Looks Like C

- Goal was to make a programming language that people would pick up easily
- There are lots of C and C++ programmers, so make it much like C
- Arithmetic & relational operators are the same: +, -, *, / and <, >, <=, >=
- Assignment is the same: a = b;
- Conditional & looping statements are the same: if/else, while, for, do, break, continue, switch
- Arrays are the same: a[i] and b[i][j]

What's Different?

- Java allows method overloading
  - C++ does this, but C does not
  - C++ also allows operator overloading; Java does not
- The Java numeric types all conform to IEEE standards
  - C numeric types can vary depending on platform
- Java does not have explicit pointers
- In Java, there is a separate String class
  - A String is *not* the same as an array of characters and it is *not* terminated by the NUL character
- Java does automatic Garbage Collection
- Many other differences…
- Java is claimed to be safer, more portable, and easier to use than C++

Object Oriented Programming

- This is a style of programming based on the ideas of
  - Objects
  - Classes
  - Inheritance
- Java is based on these ideas
- Currently, this is the *best* of known programming styles
- An *object* is a software bundle of data and related operations (the operations are called *methods* in Java)
- A *class* is a template that defines objects of a certain kind
- Using one *class*, I can create several *objects*, where each is an *instance* of this class

Simple Inheritance

- Classes can be defined in terms of other classes
  - If a new class B is based on a previous class A then
    - B is a *subclass* of A
    - A is a *superclass* of B
- In general, the variables and operations of a class are available to its subclasses
- Some classes in Java
  - String
  - Vector
  - Stack
  - Hashtable
- Stack is a *subclass* of Vector which is a *subclass* of Object
- All Java classes are *subclasses* of Object

Java Programs

- A Java program consists of a number of interacting classes
  - *All* methods and *all* variables reside within some class
- When an application runs
  - You specify a class
  - The "system" looks for and runs the method that looks like
    - public static void main(String[ ] args)

Java Programs: Applets

- When a Java Applet runs
  - The web page specifies a class
  - The “system” looks for these methods
    - public void init()
      - Runs when Applet is first loaded
    - public void start()
      - Runs when Applet appears on screen
    - public void stop()
      - Runs when Applet is off screen
    - public void destroy()
      - Runs when Applet is terminating

Object Basics

- Primitive types in Java:
  - byte, short, int, long
  - float, double
  - char
  - boolean
- *Everything* else is an Object
  - Each object is an *instance* of a Java *class*
  - There are *many* predefined Java classes
- Operators (with one exception) work only on primitive types
  - What’s the exception?
- Each Java variable holds one of two things:
  - a primitive type or
  - a *reference* to an object

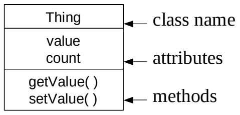
A Simple Example Class

```
public class Thing {
    private int value;
    public static int count;
    public void setValue (int v) { value = v; }
    public int getValue ( ) { return value; }
    // Plus other methods
}
```

When

```
Thing t = new Thing( );
```

is executed, an object is created that looks like this



Warning: The picture suggests that each object gets its own copy of each method. This provides some good intuition, but is not really true…

Some Terminology

```
public class Thing {
    private int value;
    public static int count;
    public void setValue (int v) { value = v; }
    public int getValue ( ) { return value; }
    // Plus other methods
}
```

- private?
- static?
- static members vs. instance members?
- function vs. procedure?
- accessor methods vs. modifier methods?

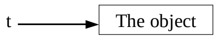
Objects vs. References

When

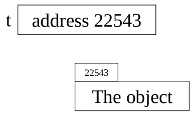
```
Thing t = new Thing ( );
```

is executed, the variable s does *not* contain the object

- Instead, it contains a *reference* to the object
- In other words, t contains the address of the place in memory where the object is stored
- Pictorially, we represent this as



- In fact, it’s more like this



Object vs. Reference Example

```
public class Thing {
    private int value;
    public static int count;
    public void setValue (int v) { value = v; }
    public int getValue ( ) { return value; }

    // A constructor
    public Thing ( ) { count++; }

    // Plus other methods
}
```

What happens?

```
Thing t1;
Thing t2;
t1 = new Thing ( );
t2 = t1;
t2.setValue(4);
System.out.println(t2.getValue( ));
t2 = new Thing ( );
System.out.println(t1.getValue( ));
System.out.println(Thing.count);
```

Null

- What happens after the declaration, but before the assignment?

```
Thing t1;
// What has happened here?
t1 = new Thing( );
```

- The variable t1 exists, but it contains no reference
  - It holds the special value *null*
  - *null* can be assigned to any object variable
  - *null* can be used in "==" tests

Equality

- The "==" operator in Java tests whether two variables contain the same value
  - For primitive types, this is what we want
  - For objects, this compares "addresses"
- Need an "equals( )" method that compares the contents of the object

What happens?

```
Thing t1 = new Thing ( );
Thing t2 = new Thing ( );
t1.setValue(44);
t2.setValue(44);
System.out.println( t1 == t2 );
```

An Improved Thing class

```
public class Thing {
    private int value;
    public static int count;
    public void setValue (int v) { value = v; }
    public int getValue ( ) { return value; }
    // A constructor
    public Thing ( ) { count++; }

    // Equality test
    public boolean equals (Thing other) {
          return value == other.value;
    }
    // Plus other methods
}
```

- Every class automatically has an equals( ) method
  - The default equals( ) method is inherited from Object
    - This is usually *not* what you actually want
    - You often need to write your own equals( )

Assignment vs. Copying (Cloning)

- What happens if we really want to make a copy of an object?
- Can't do it this way:

```
Thing t1 = new Thing( );
// Do stuff with t1; now make a copy
Thing t2 = new Thing( );
t2 = t1;
```

- Instead we use the "clone( )" method:

```
Thing t2 =  t1.clone( );
```

- Can use inherited (from Object) clone( ) if class Thing implements Cloneable

```
public class Thing {
    private int value;
    public static int count;
    public void setValue (int v) { value = v;
    }
    public int getValue ( ) { return value; }
    public Thing ( ) { count++; }
    public boolean equals (Thing other) {
          return value == other.value;
    }
    public Thing clone ( ) {
          Thing thing = new Thing( );
          thing.value = getValue( );
          return thing;
    }
}
```

Another "must-have" Method

- Methods that appear in many classes
  - equals( )
  - clone( )
  - toString( ) controls what an instance of your class looks like when printed
- All these methods have default versions that are defined in the class Object

- A toString( ) method for Thing:

```
public String toString ( ) {
      return "[ Thing " + value + "]";
}
```

Parameter Passing

- In Java, all parameters are passed by copying their values
  - For primitive types, this creates a new copy
  - For objects, this makes of copy of the object's reference

- An example "change" method

```
public void change (int j, Thing t) {
    j = 4; t.setValue(5);
}
```

- What does the following code do?

```
Thing t1 = new Thing( );
t1.setValue(1);
int i = 10;
change(i,t1);
```

- What happens if change( ) sets t to null?