CS18000: Problem Solving and Object-Oriented Programming

Methods and Classes

Video 1
Static and Non-Static Fields

Methods and Classes

Defining Methods

Parameters

Return Values

Java Class

- Java class includes…
  - Fields: attributes of an object or the class
  - Methods: operations on an object or the class
- Within certain limitations:
  Fields are accessible to methods of the class
- Fields and methods may be static or non-static
  - When static, fields and methods are "of the class"
  - When non-static, fields and methods are "of the object"

Static and Non-Static Fields

- *Static field*:
  - One memory location shared by all objects of the class
  - Same value shared by all methods
  - What is this good for?
- *Non-static field*:
  - Each instance of the class (object) has its own memory location for the field
  - Different value (in general) in each object

Java Terminology: Variables

- *Instance variables*: non-static fields in a class declaration
- *Class variables*: static fields in a class declaration
- *Local variables*: variables in method or block
- *Parameters*: variables in a method declaration

Source: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

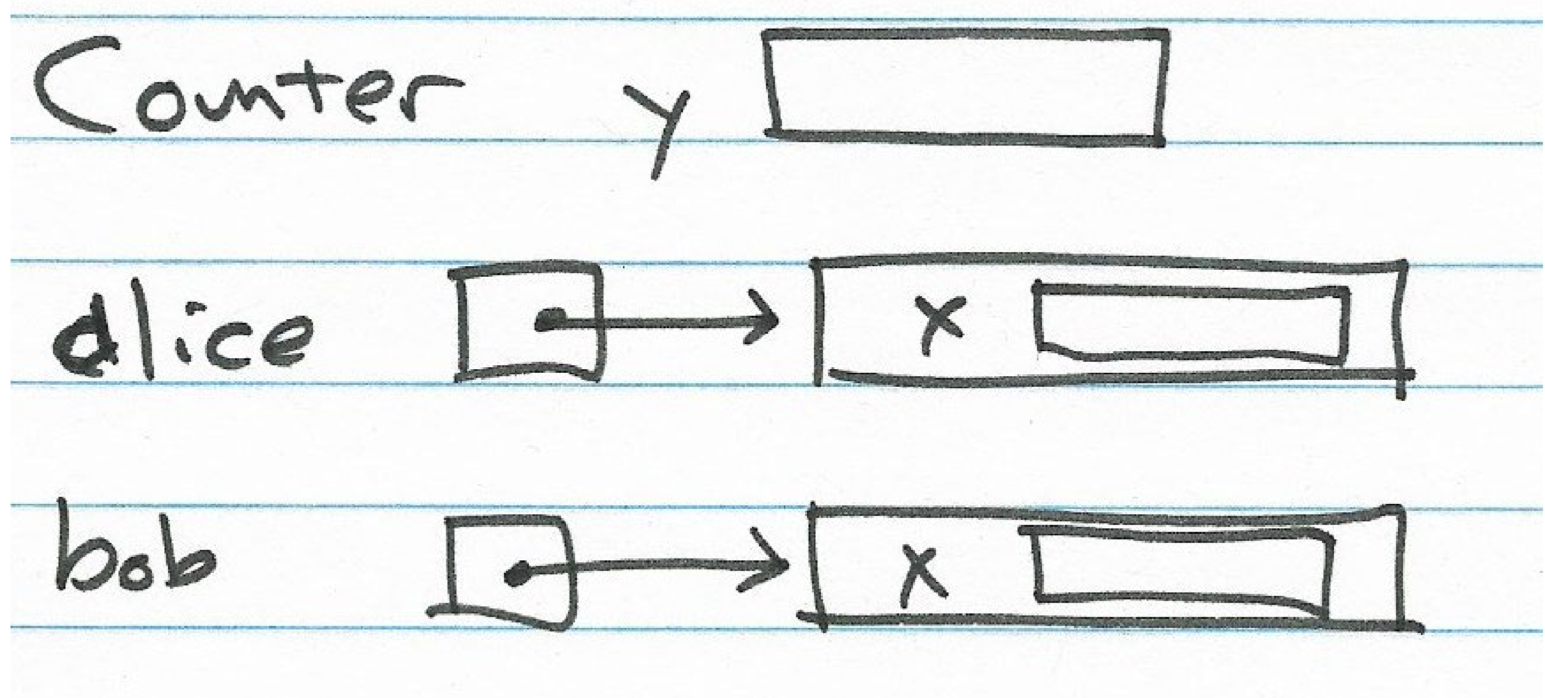
Problem: Counter

- Create a class with static and non-static counters
- Create objects
- Increment the counters
- Print values before and after

Solution 1: Counter

```
public class Counter {
    int x;
    static int y;

    public static void main(String[] args) {
        Counter alice = new Counter();
        Counter bob = new Counter();

        alice.x = 10; alice.y = 42;

        bob.x = 50; bob.y = 99;

        System.out.println(alice.x); System.out.println(alice.y);
        System.out.println(bob.x); System.out.println(bob.y);

        alice.y++;

        System.out.println(alice.y); System.out.println(bob.y);

        Counter.y++;

        System.out.println(alice.y); System.out.println(bob.y);
    }
}
```

Solution 1: Counter



Video 2
Methods

Methods

- Method: a parameterized block of code that may return a value
- Every method exists inside some class
- Useful for…
  - Reusability: Reduce redundancy in code
  - Readability: Identify logical operation by name (abstraction)
  - Modularity: software developers can code and test independently

Basic Method Syntax

```
return_type methodName(param_list) {
    statements;
    return_if_needed;
}
```

- *return_type*: type of value to be returned (`void` if no return value)
- *param_list*: list of parameters expected by method (includes types and local name, can be empty)

Parameters and Arguments

- Parameters allow a method to work on different data values
- *Parameter*: a *variable* local to a method
- *Argument*: an *expression* that is passed to the corresponding parameter at time of method call
- Argument *values* are copied into parameter *variables*
- Argument values to a method must match the number and types of parameters of the method

```
int calc (double x, int y) {… return z;}
sum = calc (r,s);
```

Parameters and Arguments

main

double r 21.46

int s 15

int sum

sum=calc(r,s);

calc(double x, int y)

double x

int y

int z

Flow of Control for Method Calls

- Before call: argument values at calling site are copied to parameter variables in called method
- Then…
  - The calling method is "suspended"
  - The called method begins at the top of the method body and runs to completion (return statement or the end)
  - The calling method continues where it left off
- After call: Return value from called method becomes value returned to calling site

Parameters: Call by Value

- Parameter variables are distinct from variables passed in as arguments

```
void changer(int x) {
    x = 12;
}
…
    int y = 5;
    changer(y);
    System.out.println(y); // prints 5
```

Parameters: Call by Value

main
int y [5]

changer (y);

changer (int x)
int x [ ]

Solution 2: Counter

```
public class Counter {
    int x;
    static int y = 42;

    Counter(int x) {
        this.x = x;
    }

    public static void main(String[] args) {
        Counter alice = new Counter(100);
        Counter jimmy = new Counter(500);

        System.out.printf("%s: x = %d, y = %d\n", "alice", alice.x, alice.y);
        System.out.printf("%s: x = %d, y = %d\n", "jimmy", jimmy.x, jimmy.y);

        alice.x++;
        alice.y++;
        jimmy.x++;
        jimmy.y++;

        System.out.printf("%s: x = %d, y = %d\n", "alice", alice.x, alice.y);
        System.out.printf("%s: x = %d, y = %d\n", "jimmy", jimmy.x, jimmy.y);
    }
}
```

Solution 3: Counter

```
public class Counter {

    // [fields and constructor omitted]

    static void display(String name, Counter c) {
        System.out.printf("%s: x = %d, y = %d\n", name, c.x, c.y);
    }

    public static void main(String[] args) {
        Counter alice = new Counter(100);
        Counter jimmy = new Counter(500);

        display("alice", alice);
        display("jimmy", jimmy);

        alice.x++;
        alice.y++;
        jimmy.x++;
        jimmy.y++;

        display("alice", alice);
        display("jimmy", jimmy);
    }
}
```

Video 3
Scope of Variables

Static and Non-Static Methods

- *Static method:*
  - May only access static fields and call other static methods in the class
  - Can be called using class name: `Math.pow(2,5)`
- *Non-static method:*
  - May also access non-static fields associated with the particular object
  - Must be associated with an object in some way in order to be called, e.g., `t3.describe()`
  - If no object specified, "`this`" implied: `describe()` is same as `this.describe()`

Extended Method Syntax

```
[static] return_type methodName(param_list)
{
    statements;
    return_if_needed;
}
```

- *static:* method is a static method
- Not *static*: method is an “instance method” (AKA “object method”)

Constructors

```
Counter c1 = new Counter(500);
```

- Not exactly a method, but very similar
- Callable using “new” operator
- May take parameters
- May access all fields of class (including static)
- *Main task: Initialize the object being allocated*
- Does not explicitly return a value…
- …but the new operator that started it returns a reference to the newly created object

Scope of Variables

- Scope: where in the code the variable is usable
- Basic rule: A variable is usable from the point of declaration to the end of the enclosing block
- Special cases…
  - Parameters: accessible within the method body
  - For loop variable: accessible within heading and body

Example: Scope1a

```
public class Scope1 {
    int x;

    void one(int x) {  // OK: hides field x
//      int x;  // cannot have in same scope as parameter
    }

    void two() {
        int x = 10;  // OK: hides field x

        while (true) {
//          int x;  // error: same scope as x above
        }
    }
}
```

Example: Scope1b

```
public class Scope1 {

    public static void main(String[] args) {
        { // independent block 1
            int x = 12;
        }

        { // independent block 2
            int x = 15;
        }

        for (int i = 0; i < 10; i++)  // i is available in header and body
            System.out.println(i);

//      System.out.println(i);  // i is now "out of scope"

}
```

Limitations Because of Scope Rules

- Cannot have two variables with same name active in same scope (compiler gives a syntax error)
- Exception:
  – parameters and local variables can “shadow” (hide) fields with the same name
  – Use `this.field` to access hidden `field`
- You’ve seen `this` used in many constructors

Example: Scope2

```
public class Scope2 {
    int x;
    String name;

    Scope2(int x, String aName) {
        this.x = x;   // common style to initialize a field
        name = aName; // alternative style that doesn't require 'this'
    }

    void doit() {
        int x = 10;  // OK: hides field x

        this.x = x; // allows access to hidden field x
    }

    public static void main(String[] args) {
        Scope2 sc1 = new Scope2(12, "Fred");
        Scope2 sc2 = new Scope2(25, "Ralph");
    }
}
```

What is `this` anyway?

- Java reserved word…
- …cannot have a variable named “this”
- Used only within a non-static method or constructor
- It is a *reference* to the “current object”
- Used to access any field or method (member) of the current object

Video 1
Designing a Class

When designing a class… (1)

- Non-static fields are the attributes of the objects of the class
  - Wheel diameter
  - Tree circumference
  - Puzzle matrix
- Static fields are shared by all objects of the class
  - Constants
  - Totals or other variables that accumulate across all objects

When designing a class… (2)

- Non-static methods operate on the attributes of the class (think: “do an action on/with object”)
  - computeDiameter
  - computeSolution
- Static methods operate on static variables or use no non-local variables at all
  - readPuzzle, readWords, computeSqrt
  - Utility methods: process parameters, return value
  - main method of a program

Problem: Modeling Trees

- Individual trees have a circumference
- Collectively, a number of trees (Tree objects) have been created (with new)

Solution: Modeling Trees

```
public class Tree {
    double circumference;
    static int numberOfTrees = 0;

    Tree(double circumference) {
        this.circumference = circumference;
        numberOfTrees = numberOfTrees + 1;
    }

    double getRadius() {
        return circumference / (2 * Math.PI);
    }

    static int getNumberOfTrees() {
        return numberOfTrees;
    }
}
```

Problem: Making Trees

- Write a program to create a number of trees
- Print how many trees were created
- Could use a counter in the method(s) that create a new Tree…
- … but using a Tree static variable keeps the bookkeeping centralized without having to insert the same code in all places that create a Tree
- Math.random() returns a number from [0.0 to 1.0)

Solution: Tracking the Trees

```
public class TreeMaker {
    public static void main(String[] args) {
        while (Math.random() < 0.9) {
            Tree t = new Tree(Math.random() * 100);
            System.out.printf("tree has radius %.3f\n",
                                    t.getRadius());
        }
        System.out.printf("created %d trees\n",
                                Tree.getNumberOfTrees());
    }
}
```

Video 2
Extent, Passing References, Overloading

Methods and Classes

Extent, Passing References, Overloading

Encapsulation

Accessors and Mutators

Extent (vs. Scope)

- Scope: Where a variable is “visible”
- Extent: How long a value is kept (its “lifetime”)
- *Variable*: lifetime same as scope
  - When block left, value of variable is lost
  - So, initializers are re-done on block entry
  - `while (…) { int i = 0; i++; System.out.println(i); }`
- *Object*: lifetime lasts until it is no longer accessible (e.g., no variables reference it)

Extent (vs. Scope)

```
String s = new String ("Purdue");
        s [ ]⟶ [ "Purdue" ]
String t = s;      t [ ]
s = new String ("Indiana");
t = new String ("Ball State");
```

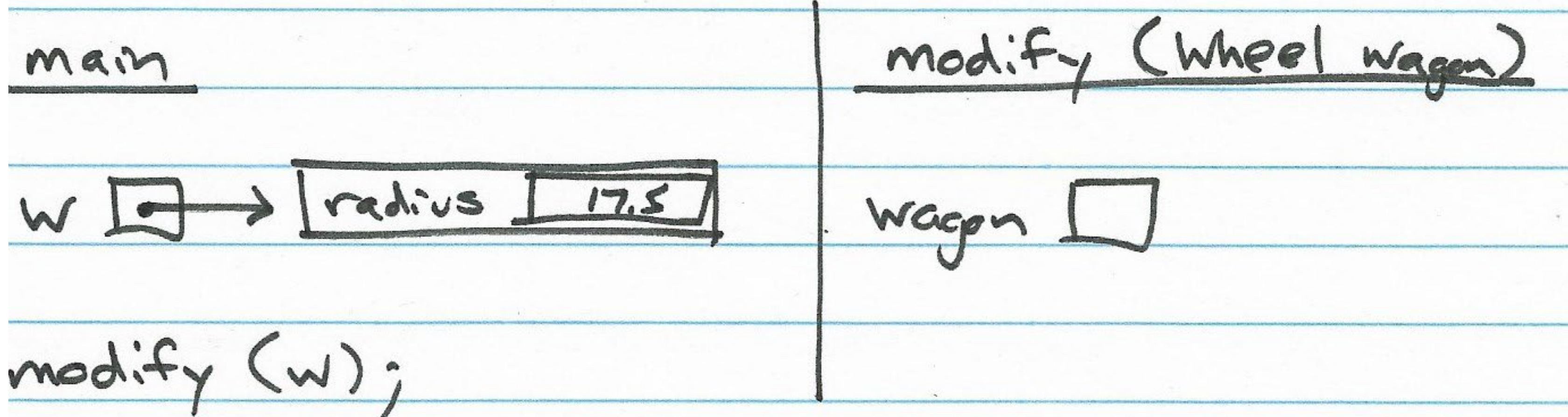
Passing References by Value

- Reminder: parameter passing is “by value”
- Passing an object reference by value means…
  - The “value” is the reference to (address of) the object
  - The called method cannot modify the variable where the reference is stored
  - But, it can modify the object it references

Example: Reference by Value

```
public class Danger {
    static void modify(Wheel wagon) {
        wagon.radius = 22.6; // modifies object referenced by wagon
    }

    public static void main(String[] args) {
        Wheel w = new Wheel (17.5);

        System.out.println(w.getRadius());

        modify(w);

        System.out.println(w.getRadius());
    }
}
```

Example: Reference by Value



Overloading Constructors and Methods (1)

- Term *signature* refers to the name and parameter types of a method or constructor

```
double getPayment (int months, double interestRate)
```

- Each constructor and method in a class must have a unique signature
- Same names are OK, but parameter types must be different

```
double getPayment (int years)
double getPayment ()
double getPayment (int owner, String address)
```

Overloading Constructors and Methods (2)

- Java matches argument types with parameter types to choose the right constructor or method to invoke
- Called *overloading*: we're overloading the meaning of the method name
- Many built-in classes use constructor and method overloading (see `println` in http://docs.oracle.com/javase/6/docs/api/java/io/PrintStream.html)

Example: Improving isPalindrome

```
public class Palindrome {
    static boolean isPalindrome(String s) {
        if (s == null || s.length() <= 1)
            return true;

        while (s.length() > 1) {
            char first = s.charAt(0);
            char last = s.charAt(s.length() - 1);
            if (first != last)
                return false;
            s = s.substring(1, s.length() - 1);
        }
        return true;
    }

    static boolean isPalindrome(int x) {
        return isPalindrome(Integer.toString(x));
    }
}
```

Video 3
this() in Constructors

Special Trick: `this()` in Constructor

- Use this(…) to invoke one constructor from another
- Must be first line in current constructor
- Java matches argument types to determine which constructor to call
- Other constructor returns to calling constructor

Example: PurdueStudent (1)

```
public class PurdueStudent {
    boolean hasName;
    String name;
    int puid;

    // constructor with int...
    PurdueStudent(int puid) {
        this.puid = puid;
        hasName = false;
    }

    // constructor with String and int...
    PurdueStudent(String name, int puid) {
        this(puid);
        this.name = name;
        hasName = true;
    }

    // [see next slide…]
}
```

Example: PurdueStudent (2)

```
public class PurdueStudent {
    // [see previous slide…]

    void printStudent() {
        if (hasName)
            System.out.println(name + ": " + puid);
        else
            System.out.println("(no name): " + puid);
    }

    public static void main(String[] args) {
        // call int-only constructor...
        PurdueStudent p1 = new PurdueStudent(1010337138);

        // call String-int constructor...
        PurdueStudent p2 = new PurdueStudent("Drake", 1123441245);

        p1.printStudent();
        p2.printStudent();
    }
}
```

Example: Overloader

```
public class Overloader {
    int calc (double x, int y) {
    ...
    return z;}
// valid overload, 3 parameters
    int calc (double x, int y, int z) {
    ...
    return z;}
// valid overload, 1 parameter
    double calc (double x) {
    ...
    return q;}
// valid overload, 2 parameters, different types
    int calc (double x, String s) {
    ...
    return z;}
// INVALID overload, return type is not part of the signature
    double calc (double a, int b) {
    ...
    return q;}
    }
}
```

Video 4
Encapsulation

Encapsulation

- Java supports team work through encapsulation
- A form of information hiding or "need to know"
- Encapsulation serves two purposes
  - Hides implementation details from other programmers, allowing the author to make changes without affecting other programmers
  - Prevents other programmers from modifying certain fields or calling certain methods that might leave the object in an inconsistent or unexpected state

Java Access Modifiers

- Can apply to members: fields and methods
- Modifiers control access to members from methods in other classes
- This list is from least to most restrictive:

Keyword	**Restriction**
`public`	None (any other method can access)
`protected`	Only methods in the class, subclasses, or in classes in the same package can access
[none]	Only methods in the class or in classes in the same package can access (called “package private”)
`private`	Only methods in the class can access

Conventional Wisdom

- Make methods public
  - Allows anyone to use them
  - They should be written defensively to “protect” the object internal state (i.e., attribute fields)
- Make fields private
  - Keeps them safe from unexpected changes
  - Only your methods can modify them
- Constants (“`final`” fields) can be made public since they can’t be changed anyway

Accessor and Mutator Methods

- With fields being private, all access to them from outside the class is via methods
- There is no special Java syntax, just naming convention:
  - Accessor: “get…” access (read) field in an object
  - Mutator: “set…” mutate (change) field in an object
- Accessor methods allow read-only access
- Mutator methods allow “controlled” change

Better Wheel Class

```
public class Wheel {
    private double radius;
    public Wheel(double radius) {
        this.radius = radius;
    }
    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
    public double getArea() {
        return Math.PI * radius * radius;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double r) {
        if (r>0 && r<=1000)
            radius=r;
    }
}
```

Using Accessor and Mutator Methods

```
public class Transportation {

    public static void main(String[] args) {
        Wheel w = new Wheel (17.5);

        double circ = w.getCircumference();

        double rad = w.radius;  // not allowed
        double rad = w.getRadius();

        w.radius = 15.4; // not allowed
        w.setRadius(15.4);
     }
}
```