Course Overview (From Lecture 1)

What is CS112?
- A broad, programming-centric introduction to computer science

Goals
- Demystify computer systems
- Empower you to exploit available technology
- Build awareness of substantial intellectual underpinnings

Topics
- Programming in Java
- Key program design techniques & problem-solving skills
- Programming tools & important libraries and data structures
- Applications to science, engineering, and commercial computing

"Computers are incredibly fast, accurate, and stupid; humans are incredibly slow, inaccurate, and brilliant; together they are powerful beyond imagination." — Albert Einstein

Final Review

Final Exam: 2:00pm - 4:00pm, DAVIS AUD
Tuesday, May 1st, 2012,

For more info about the final exam, please see:

http://zoo.cs.yale.edu/classes/cs112/lectures/exam2

Topic: Static/Instance Methods/Variables: I am confused
Static Variable

- A variable declared with the `static` modifier
  - Behavior:
    - has a single copy across the whole program
    - is created and initialized when the program starts
  - Why (example usage)?
    - class-wide counters to keep track the number of objects created from the class
    - single instance object, e.g., `out` is a static variable in the `System` class

Static Methods

- Intention: reflect common class behaviors, not dependent on any instance of objects
  - a static method can be either public or private; the order of the modifiers such as public/private/static can be interchanged, but by convention visibility modifiers come first
- Invocation:
  - inside class definition: just call the method name
  - outside class definition: `ClassName.methodName(…)`

Static Method Invocation

```java
class Tester {
    public static int triple (int num) {
        int result;
        result = num * 3;
        return result;
    }
    ...

    value = Tester.triple (5);
    // It is possible to invoke a static method through an object:
    Tester t = new Tester();
    value = t.triple (5);
}
```

Instance Methods

- Intention
  - reflect the behaviors of objects created from the class, depends on individual objects’ states
  - instance methods can be accessed only through an object:
    - `ClassName objVar = new ClassName(...);
      objVar.methodName(...);`
  - can access both instance and class/static variables
Scoping and Access Rule

- Class-scope variables
  - static variables: can be accessed (in scope) in all methods in the class; outside the class, depends on public/private modifier (ObjectName.staticVarName)
  - instance variable: can be accessed (in scope) in instance methods defined in the class; outside the class, depends on public/private modifier (objVarName.varName)

- Block-scope variables
  - can be accessed in the enclosing block
  - a local variable (method parameter) can shadow a class-level variable with the same name
    - Use this to access the shadowed class-scope variable

Class-Scope Variables and Methods in a Class

<table>
<thead>
<tr>
<th></th>
<th>static/class method</th>
<th>instance method</th>
</tr>
</thead>
<tbody>
<tr>
<td>static/class variable</td>
<td>can access</td>
<td>can access</td>
</tr>
<tr>
<td>instance variable</td>
<td>cannot access</td>
<td>can access</td>
</tr>
</tbody>
</table>

Accessibility of variables in methods defined in the same class

Basic Rule: Calling a Method

- Each time a method is called, each actual argument in the invocation is copied into the corresponding formal argument
  - if a value type, then it is the value that is copied
  - if a reference type, then it is the reference that is copied
    - Objects and arrays are reference types
  - The formal argument and the actual argument are different variables, with different memory locations, even if they have the same name
**Calling a Method: Value**

```java
calling a method: value

public int SquareSum (int num1, int num2)
{
    num1 = num1 + num2;
    return num1 * num1;
}

t = SquareSum (num1, num2);
t = 3;
t = 2;
```

**Calling a Method: Reference**

```java
calling a method: reference

Account bobAcct = new Account("Bob", 22222, 200.0);
double balance = 0;
doubleBalance (bobAcct);

static void doubleBalance(Account act)
{
    double balance = act.getBalance();
    act.setBalance(balance * 2);
}

doubleBalance (bobAcct);
```

**Example: Parameter Passing**

```java
example: parameter passing

public class Num {
    private int value;
    public Num(int update) {
        value = update;
    }
    public void setValue(int update) {
        value = update;
    }
    public String toString() {
        return value + "\n";
    }
}
```

**Tracing the Parameters:**

```java
tracing the parameters:

public static void main(String[] args) {
    int a1 = 111;
    Num a2 = new Num(222);
    Num a3 = new Num(333);
    changeValues(a1, a2, a3);
}

public void changeValue(int f1, Num f2, Num f3) {
    f1 = 999;
    f2.setValue(888);
    f3 = new Num(777);
}
```
Tracing the Parameters: `In tester.changeValues(a1, a2, a3)`

```java
public static void main(String[] args) {
    int a1 = 111;
    Num a2 = new Num(222);
    Num a3 = new Num(333);
    changeValues(a1, a2, a3);
}
```

Tracing the Parameters: `f1=999`

```java
public void changeValue(int f1, Num f2, Num f3) {
    f1 = 999;
    f2.setValue(888);
    f3 = new Num(777);
}
```

Tracing the Parameters: `f2.setValue(888)`

```java
public void changeValue(int f1, Num f2, Num f3) {
    f1 = 999;
    f2.setValue(888);
    f3 = new Num(777);
}
```

Tracing the Parameters: `f3 = new Num(777)`

```java
public void changeValue(int f1, Num f2, Num f3) {
    f1 = 999;
    f2.setValue(888);
    f3 = new Num(777);
}
```
public static void main(String[] args) {
    int a1 = 111;
    Num a2 = new Num(222);
    Num a3 = new Num(333);
    changeValues(a1, a2, a3);
}

public void changeValue(int f1, Num f2, Num f3) {
    f1 = 999;
    f2.setValue(888);
    f3 = new Num(777);
}

Basic Rule

- Java is dynamic binding: the method invoked is always the type of the object, not the type of the variable

Exercise: A Polymorphism Problem

- Suppose that the following four classes have been declared:

```java
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }
    public void method2() {
        System.out.println("foo 2");
    }
    public String toString() {
        return "foo";
    }
}

class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}

class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }
    public String toString() {
        return "baz";
    }
}

class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```
A Polymorphism Problem

What would be the output of the following client code?

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Finding output with tables:

<table>
<thead>
<tr>
<th>Method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td></td>
<td>baz 1</td>
<td></td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td></td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td></td>
<td>baz</td>
<td></td>
</tr>
</tbody>
</table>

Finding output with tables:

<table>
<thead>
<tr>
<th>Method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td>foo 1</td>
<td>baz 1</td>
<td>baz 1</td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td>foo 2</td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td>foo</td>
<td>baz</td>
<td>baz</td>
</tr>
</tbody>
</table>
Use Diagramming

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.

Exercise 2:

- The methods sometimes call other methods (tricky!).

```java
public class Ham {
    public void a() {
        System.out.print("Ham a ");
        b(); // whose b()? Ham's?
    }
    public void b() {
        System.out.print("Ham b ");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b ");
    }
    public String toString() {
        return "Yam";
    }
}
```

Polymorphism Answer

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.print(pity[i].toString());
    if (i == 0 || i == pity.length - 1) {
        System.out.println();
    }
}
```

- Output:
  - baz
  - baz 1
  - foo 2
  - foo
  - foo 1
  - bar 2
  - baz
  - baz 1
  - mumble 2
  - foo
  - foo 2
  - foo 2

Another problem 2

```java
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b ");
    }
}
```

- What would be the output of the following client code?

```java
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i].toString());
    food[i].a();
    System.out.println();
    food[i].b();
    System.out.println();
}```
The table

<table>
<thead>
<tr>
<th>method</th>
<th>Ham</th>
<th>Lamb</th>
<th>Yam</th>
<th>Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Ham a</td>
<td></td>
<td>Yam a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b()</td>
<td></td>
<td>super.a()</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Ham b</td>
<td>Lamb b</td>
<td></td>
<td>Spam b</td>
</tr>
<tr>
<td>toString</td>
<td>Ham</td>
<td></td>
<td>Yam</td>
<td></td>
</tr>
</tbody>
</table>

The answer

```java
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
  System.out.println(food[i]);
  food[i].a();
  food[i].b();
  System.out.println();
}
```

- Output:
  - Ham
  - Ham a
  - Lamb b
  - Spam b
  - Ham
  - Ham a
  - Lamb b
  - Spam b
  - Ham
  - Yam a
  - Spam b
  - Lamb b
Common Questions We Saw: OO

- The role of "this" and "super" keywords
- Correctly overriding methods in OOP
- Benefits/implications of the "static" keyword, particularly in an OOP context