Classes

CS 99 – Summer 2000
Michael Clarkson
Lecture 7

Administration

- Lab 7 due tomorrow
  - Question: `"Lab 6".equals("SquareRoot")`?
- Lab 8 posted today
- Prelim 2 in six days!
  - Covers two weeks of material:
    - lectures 5-7
    - labs 5-8
  - Review: Monday, 7:30-8:30pm, Philips 203
  - Exam: Tuesday, in class
    - Upson 215 (10:00)
    - Upson B17 (11:30)

Agenda

- Objects Part II
- Command Line Arguments
- Classes

Objects Part II

- Object aliases
- Objects as arguments

Review: Object References

- Recall that declaring an object and creating (instantiating) it are two separate steps
- We declare a reference, and that reference refers to an object:

```
Car myCar;
```

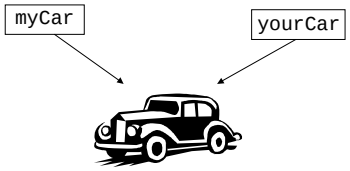

Object Aliases

- More than one reference can refer to the same object, e.g.:

```
Car myCar, yourCar;
myCar = new Car();
yourCar = myCar;
```

- This is called *aliasing*

Object Aliases [2]



Object Aliases [3]

- Anything that happens to an alias, happens to the real object
- So:

```
yourCar.honkHorn();
```

Honks my car's horn!

Object Equality

- References are why the `==` operator doesn't work like you might suspect on objects
- `==` returns whether the references are equal, not whether the data contained by the objects are equal
- In general, you have to use the `equals` method of an object to determine equality

Objects as Arguments

- Objects can be used as arguments to methods
- This is nothing new:

```
static void System.out.println(String s)
```

- There is a difference between passing primitive types and objects to methods

Objects as Arguments [2]

```
public static void main(String[] args) {
  int a = 0;
  v1(a);
  System.out.println(a);
  String s = "Hello";
  v2(s);
  System.out.println(s);
}

static void v1(int i) { i++; }
static void v2(String s) { s="Goodbye" }
```


Calling Semantics

- *Call by value*:
  - changes to value are not reflected outside of method
  - used for primitive types
  - used for objects (i.e., references)
    - reference itself is not changed, BUT…
    - changes to object *are* reflected outside of method

Objects as Arguments [3]

```
class Car {
   String color = "Blue";
}

public static void main(String[] args) {
   Car car = new Car();
   repaint(car);
   System.out.println(car.color);
}

static void repaint(Car car) {
   car.color = "Red";
}
```


Command Line Arguments

- What Metrowerks is hiding from you
- More of `main` revealed

What Metrowerks is Hiding

- Without an IDE like Metrowerks, you compile a Java program on the command line:

```
C:\> javac HelloWorld.java
```

- That produces a file named HelloWorld.class that you then run by typing:

```
C:\> java HelloWorld
```


Running Java Programs

- You can also provide more information than the name of the program:

```
C:\> java ReverseWords red green
der neerg
```

- Words after the name of the program are called *command line arguments*
- Another way of getting input to a program

`main`

```
public static void main(String[] args)
```

- Command line arguments are passed to your main method as an array of Strings
  - Each word after the name of the program becomes one element of the array
- So given:

```
java ReverseWords red green
```

`args[0]` is "red"
`args[1]` is "green"

Classes

- Class anatomy
  - Fields
  - Methods
  - Files
- `this`
- Accessing fields and methods
- Constructors
- Instance data
- Modifiers
  - `public`, `private`, `static`

Class Anatomy

- Recall that classes have two types of *members*:
  - Fields
  - Methods

```
class Car {
   int speed;
   int numDoors;

   void accelerate(int amount) {
       ...
   }
}
```


Class Anatomy [2]

- Classes are usually defined in separate .java files
  - Files have the same name as the class they contain
- So for labs where you create classes, you'll have several .java files to turn in

Scope of Fields

- Fields have scope throughout the entire class
- Can be used in any method of the class by writing their name, e.g.:

```
void accelerate(int amount) {
      speed += amount;
}
```

- Can be used outside of the class by preceding the field name with an object reference, e.g.:

```
Car myCar;
myCar.speed = 55;
```


Invoking Methods

- To call a method from outside of the class, use an object reference, e.g.:

```
myCar.turnLeft();
```

- To call a method of a class from inside the same class, no reference is needed, e.g.:

```
void turnLeft() {
      accelerate(-10);  // slow down
      ...
}
```


Constructors

- *Constructors* are special methods defined inside a class
- A constructor is called whenever an object of the class is created
- Constructors are used to initialize an object's fields properly
- Also used for convenience

Constructors [2]

- A constructor has
  - the same name as the class it is in
  - no return type
  - whatever parameter list you want

```
class Point {
   double x, y;
   Point(int initX, int initY) {
       x = initX;
       y = initY;
   }
}
Point p = new Point(1, 2);
```

Constructors [3]

- Very simple constructors don't even have any parameters – *default constructor*

```
class RentalCar {
  int gasLeft;
  RentalCar() {
      gasLeft = FULL_TANK;
  }
}
RentalCar car = new RentalCar();
```


Constructors [4]

- You can include as many constructors as you want inside a class
- Java picks the right one to call based on the types of the arguments you use when constructing an object
- If you provide no constructors, Java automatically provides a *default constructor* that takes no arguments and basically does nothing

Instance Data

- The data contained by one particular object is called *instance data*
- So for `Point`s, the x and y coordinates are instance data
- Each object has its own separate memory reserved for its instance data
- It can be different than the data contained by any other object of the same class

Instance Data [2]

- Instance data is not shared between objects

```
Point p1 = new Point(0, 0);
Point p2 = new Point(-1, 4);

p1.x != p2.x
p1.y != p2.y
```


Instance Data [3]

- Objects are responsible for managing their own instance data
- Users of the object usually aren't allowed access to instance data, except through methods defined by the object's class
- Objects are self-governing

Visibility Modifiers

- Visibility modifiers are keywords used in declaring members
- They make members either visible or invisible from outside the class
- `public`
  - A public member is accessible outside of the class using the *`reference.member`* syntax
- `private`
  - A private member is not accessible outside of the class
  - It is accessible from inside any method of the class

toString()

- One method that is common to have in many classes is `toString()`
- It is called whenever an object of the class is used in a concatenation operation

```
public String toString() {
  return "(" + x + "," + y + ")";
}

System.out.println(p1);
```


main

```
public static void main(String[] args)
```

- `main` is `public` to make it accessible from outside the class it is declared in
- Otherwise, Java couldn't call your main method

Publicity Guidelines

- Usually, all fields are private
- Public methods are for users of the class
- Private methods are called from public methods to help implement them
  - But aren't meant for users of the class to call

static

- `static` is a modifier
- Indicates that the member is shared by *all* instances of a class
- For fields:
  - the data is shared by all objects
  - if one object changes it, it's changed for every object
  - best use is for class-wide constants (`static final`)

static Methods

- Static methods have no associated object
- No instance data involved
- Not invoked using the `reference.method()` syntax
- Invoked with `Class.method()` syntax

static Methods [2]

- Examples:
  - `Math.sqrt()`
  - `Console.readInt()`
- Since they have no associated object, not allowed to use instance variables

main

```
public static void main(String[] args)
```

- `main` is `static` because
  - it belongs to no particular object
  - belongs to entire program
- We now know what every part of `main` means!

System

- `System` is a class that contains useful methods for working with the computer system on which a program runs
- `out` is a `static` field of `System` with the type `PrintStream`, e.g.:

```
class System {
   public static PrintStream out;
   ...
}
```


println()

- So when we invoke the println method, we're saying:
  - In the `System` class,
  - Get the `out` object, and
  - Invoke the `println()` method on it

Next Time

- Some theory behind object-oriented programming
- Advanced usage of classes