Multidimensional Arrays

Just arrays of arrays:

```
int[][] a = new int[3][];

a[0] = { 1, 2, 3 };
a[1] = null;
a[2] = { 4, 5 };

int[][] b = { { 1, 2, 3 }, null, { 4, 5 } };

System.out.println(a[2][0]);     // prints 3
```


Memory Model

All objects are in heap.

```
Date now = new Date();
Date alsoNow = now;
Date nowAgain = now.clone();

// compare pointers
boolean comp1 = (now == alsoNow);
boolean comp2 = (now == nowAgain);

// compare objects
boolean comp3 = (now.equals(alsoNow));
boolean comp4 = (now.equals(nowAgain));
boolean comp5 = (now.equals(new Date()));
```


Subclasses

Given the following class definitions

```
public class Bird
{
   public String tweat() { return "tweat"; }
}

public class Owl extends Bird
{
   public String tweat() { return "hoot"; }
   public String bob() { return "bob bob; }
}

public class Dog
{
   public String bark() { return "woof"; }
}
```


Type Sysyem

Java is statically and dynamically typed (C++ is only statically typed).

Consider: `Bird bird = new Owl();`

Static: each variable has a type.

- A variable of type `Bird` can only reference descendents of `Bird`.
- Only methods defined on `Bird`s can be invoked via a `Bird` variable. Legal: `bird.tweet()` Illegal: `bird.bob()`

Type System (cont)

Dynamic: Each object knows what type it really is.

- So `bird.getClass()` returns the `Class` object for `Owl`.
- Type casts tell compiler that object is really a subclass:

```
Owl owl = (Owl)bird;
owl.bob();
```

- Above cast only works if `bird` is really a reference to an `Owl` or a descendent of `Owl`. It is a runtime exception otherwise. E.g. `(Owl)(new Bird())`.

Type System, E.g. (cont)

What are the results of each line?
Which lines cause compile-time or run-time errors?

```
Bird b1 = new Owl();
Bird b2 = new Bird();

b1.tweet();
b2.tweet();
b1.bob();

Owl o0 = b1;
Owl o1 = (Owl)b1;
Owl o2 = (Owl)b2;
Bird b3 = (Bird)(new Dog());
```


Static/Dynamic Type

The static type of an expression is the type the compiler associates with the expression.

The dymamic type of an expression is actual type of the value of the expression when the program is running.

Assignment is allowed if the static type of RHS is a descendent of the static type of LHS: `Bird bird = new Owl();`

The static type of a type-cast expression is the type of the cast.

A cast succeeds if dynamic type of expression is a descendent of the type of the cast. `(Owl)bird;`

Operator `instanceof`

Consider

```
public String animate(Bird b)
{
   String out = b.tweet();
   if (b instanceof Owl)
      out = out + ", " + ((Owl)b).bob();
}
```

Functions, Methods, etc.

instance variables or member fields: Each instance of an object has its own copies.

member functions or methods: Functions invoked on instances of objects. Implicit `this` reference in method body refers to the object that the method was invoked on. `bird.tweet();`

Static variables or class variables: One per class. E.g. `System.out` or `Integer.MAX_VALUE` .

Static functions or class functions: No implicit `this` reference. Invoked like `<class name>.<function name>`. E.g. `Integer.parseInt("500")`

Static functions and static variables are declared with `static` modifier.

Methods: Scoping

Scoping:

- Parameter variable names shadow instance variable names. Use `this` to access instance variables.
- Superclass instance variables (and methods) are shadowed by subclass instance variables (and methods). Use `super` to access superclass variables.

Methods: Overloading

Overloading:

- Operators cannot be overloaded. I.e. `*` can only multiply numbers.
- Methods and static functions can be overloaded. The signature of a function is a combination of its name and its argument types.
- Overloading uses static types not dynamic types of arguments.

Methods: Overriding

Overriding:

- In C++ terms, all Java methods are virtual: All method invocations are dynamically dispached.
- A method in a subclass overrides same-signature method in superclass.
- Dangerous to combine overloading with overriding.

Overriding & Overloading (E.g.)

```
public class Bird
{
   public String tweat() { return "tweat"; }
   public String peck(Bird b)
      { return "A bird pecks a bird"; }
   public String peck(Owl o)
      { return "A bird pecks an owl"; }
}

public class Owl extends Bird
{
   public String tweat() { return "hoot"; }
   public String bob() { return "bob bob"; }
   public String peck(Bird b)
      { return "An owl pecks a bird"; }
   public String peck(Owl o)
      { return "An owl pecks an owl"; }
}
```


Overriding & Overloading (cont)

What are the results of?

```
Bird b1 = new Bird();
Bird b2 = new Bird();
Bird bo1 = new Owl();
Bird bo2 = new Owl();
Owl o1 = new Owl();
Owl o2 = new Owl();


b1.peck(b2);                     bo1.peck(b2);
b1.peck(bo2);                    bo1.peck(bo2);
b1.peck(o2);                     bo1.peck(o2);

o1.peck(b2);
o1.peck(bo2);
o1.peck(o2);
```


Subclassing

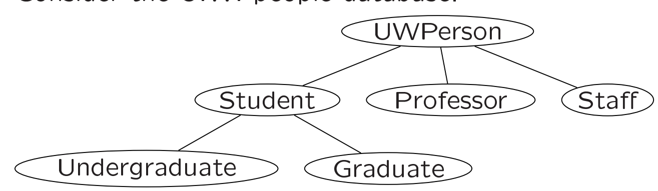
Use `extends` keyword.

Subclasses must define their own constructors. If not, a default one will be used. If any constructor is defined, the default constructor will not be used.

A class can have only one parent class – no multipule inheritance.

Multipule Inheritance?

Consider the U.W. people database:



Who teaches classes/sections? Graduates and Professors do:

```
public class Graduate extends Student
{  ...
   public void teach(Course c) {...}
}
public class Professor extends UWPerson
{  ...
   public void teach(Course c) {...}
}
```

Multipule Ineritance (cont)

How can we register teachers to teach courses?

```
public class Course
{
   ...
   public void registerTeacher(UWPerson p)
   {
      p.teach(this);
   }
}
```

Whats wrong?

Multipule Ineritance (cont)

Second try:

```
public void registerTeacher(UWPerson p)
{
   if (p instanceof Professor)
   {
      Professor prof = (Professor)p;
      prof.teach(this);
   }
   else if (p instanceof Graduate)
      ((Graduate)p).teach(this);
}
```

Works, but clunky.

Interfaces

While multipule-inheritance might be a bad idea, but the idea of having variable for objects that have the `teach(Course)` method is useful.

Solution: interfaces

A java class can implement any number of interfaces.

```
public interface Teacher
{
   public void teach(Course c);
}
```

Any class that implements this interface can be put in a `Teacher` variable.

Any class that implements `Teacher` must define its own `teach(Course)` function.

Interfaces (E.g.)

```
public class Graduate extends Student implements Teacher
{
   ...
   public void teach(Course c) {...}
}

public class Professor extends UWPerson implements Teacher
{
   ...
   public void teach(Course c) {...}
}
```

registerTeacher

Now we can define registerTeacher as:

```
public class Course
{
   ...
   public void registerTeacher(Teacher t)
   {
      t.teach(this);
   }
}
```

It can be called as:

```
Course c1 = ...;
Course c2 = ...;
Professor p = ...;
Graduate g = ...;

c1.registerTeacher(p);
c2.registerTeacher(g);
```


Iterator Interface

Consider java.util.Enumeration:

```
public interface Enumeration
{
   public boolean hasMoreElements();
   public Object nextElement();
}
```


Polymorphism

Interfaces provide means of polymorphism.

A polymorphic function is one that can by applied to arguments of different types:

```
public static printEnumeration(Enumeration e)
{
   while (e.hasMoreElements())
      System.out.println(e.nextElement());
}
```

java.util.StringTokenizer implements Enumeration:

```
StringTokenizer st = new StringTokenizer("this is a test");
printEnumeration(st);
```

You can get an Enumeration from a java.util.Vector:

```
Vector v = ...;
printEnumeration(v.enumeration());
```


IntegerRange Class

```
public class IntegerRange implements Enumeration
{
   int min; int max;
   public IntegerRange(int minIn, int maxIn)
   {
      min = minIn;
      max = maxIn;
   }
   public boolean hasMoreElements()
   {
      return min < max;
   }
   public Object nextElement()
   {
      return new Integer(min++);
   }
}
```