Method Invocation

- Syntax
  
  `o.m(arg1, arg2, ..., argn);`
  
  - Run the m method of object o with arguments arg1...argn

- Two ways to reuse method names:
  - Methods can be overridden
  - Methods can also be overloaded

Overriding

- Define a method also defined by a superclass

  ```
  class Parent {
    int cost;
    void add(int x) {
      cost += x;
    }
  }

  class Child extends Parent {
    void add(int x) {
      if (x > 0) cost += x;
    }
  }
  ```

Overriding (cont’d)

- Method with same name and argument types in child class overrides method in parent class
- Arguments and result types must be identical
  - otherwise you are overloading the method (e.g., equals from last time)
- Must raise the same or fewer exceptions
- Can override/hide instance variables
  - both variables will exist, but don’t do it

Method Dispatch

- Let B be a subclass of A, and suppose we have
  ```
  A a = new B();
  ```

- Then
  - instance methods invoked on a will get the methods for class B (in C++, virtual functions)
  - class (static) methods invoked on a will get the methods for class A
  - invoking class methods on objects strongly discouraged

Administrivia

- See me after class if you’re on the wait list
- Reading:
  - Thinking in Java, Chapters 4 and 7
Simple Dynamic Dispatch Example

```java
public class A {
    String f() {return "A.f()"; }
    static String g() {return "A.g()"; }
}

public class B extends A {
    String f() {return "B.f()"; }
    static String g() {return "B.g()"; }
    public static void main(String args[]) {
        A a = new B();
        B b = new B();
        System.out.println(a.f() + a.g() + b.f() + b.g());
    }
}
```

Overloading

• Methods with the same name, but different parameters (count or types) are overloaded

```java
class Parent {
    int cost;
    void add (int x) {
        cost += x;
    }
    void add (String s) throws NumberFormatException {
        cost += Integer.parseInt(s);
    }
}
```

Overloading (cont’d)

• method invoked is determined by both its name and the types of the parameters
  • Not return value

• resolved at compile-time, based on compile-time types

Self reference

• this refers to the object the method is invoked on
• super refers to the same object as this
  – but used to access methods/variables in superclass

Source code for classes

```java
class A {
    String f(A x) { return "A.f(A)"; }
    String f(B x) { return "A.f(B)"; }
    static String g(A x) { return "A.g(A)"; }
    static String g(B x) { return "A.g(B)"; }
    String h = "A.h";
    String getH() { return "A.getH(): " + h; }
}
class B extends A {
    String f(A x) { return "B.f(A)/ " + super.f(x); }
    String f(B x) { return "B.f(B)/ " + super.f(x); }
    static String g(A x) { return "B.g(A)"; }
    static String g(B x) { return "B.g(B)"; }
    String h = "B.h";
    String getH() { return "B.getH(): " + h + "/" + super.h; }
}
```
A a = new A();  A ab = new B();  B b = new B();
System.out.println(a.f(a) + a.f(ab) + a.f(b));
// A.f(A)  A.f(A)  A.f(B)
System.out.println(ab.f(a) + ab.f(ab) + ab.f(b));
// B.f(A)  A.f(A)  B.f(B)
System.out.println(b.f(a) + b.f(ab) + b.f(b));
// B.f(A)  B.f(A)  B.f(B)
System.out.println(a.g(a) + a.g(ab) + a.g(b));
// A.g(A)  A.g(A)  A.g(B)
System.out.println(ab.g(a) + ab.g(ab) + ab.g(b));
// A.g(A)  A.g(A)  A.g(B)
System.out.println(b.g(a) + b.g(ab) + b.g(b));
// B.g(A)  B.g(A)  B.g(B)
System.out.println(a.h + " "+ a.getH());
// A.h  A.getH():A.h
System.out.println(ab.h + " "+ ab.getH());
// A.h  B.getH():B.h/A.h
System.out.println(b.h + " "+ b.getH());
// B.h  B.getH():B.h/A.h

What to notice

• Invoking ab.f(ab) invokes B.f(A)
  – run-time type of object determines method invoked
  – compile-time type of arguments used
• ab.h gives the A version of h
• ab.getH() gives B version of h
• Use of super in class B to reach A version of methods/variables

• super not allowed in static methods

Constructors

• Declaration syntax same as C++
  – no return type specified
  – method name same as class
• First statement can be this(args) or super(args)
  – if those are omitted, super() is called
    – must be very first statement, even before variable declarations
• not used for type conversions or assignments
• void constructor generated if no constructors given

Garbage collection

• Objects that are no longer accessible can be garbage collected
• Method void finalize() called when an object is collected
  – best to avoid using it, since no way to tell when it will get called
• Garbage collection often not a major performance bottleneck
  – new/delete in C++ can be expensive too

Finalizer Example

public class FileInputStream extends InputStream
{
  private FileDescriptor fd;
  ...
  protected void finalize() throws IOException {
    if (fd != null) {
      if (fd != fd.in) { /* Don’t close standard input */
        close();
      }
    }
  }
}

Interfaces

• An interface lists supported methods
  – No constructors or implementations allowed
  – Can have final static variables
• A class can implement (be a subtype of) one or more interfaces
• Using the name of an interface as a type (i.e., to declare a variable) means
  – a reference to any instance of a class that implements the interface is a permitted value
  – null is also allowed

CMSC 433, Jeff Foster, U. Maryland (via M. Hicks, W. Pugh, A. Sussman, and A, Porter)
**Interface example**

```java
public interface Comparable {
    public int compareTo(Object o);
}

public class Util {
    public static void sort(Comparable []) {
        ...
    }
}

public class Choices implements Comparable {
    public int compareTo(Object o) {
        return ...;
    }
    Choices [] options = ...;
    Util.sort(options);
    ...
}
```

**No multiple inheritance**

- A class type can be a subtype of many other types (**implements**)
- But can only inherit method implementations from one superclass (**extends**)
- Not a big deal
  - multiple inheritance rarely, if ever, necessary and often badly used
- And it’s complicated to implement well

**Poor man’s polymorphism**

- Every object is an **Object**
- Thus, a data structure **Set** that implements sets of **Objects**
  - can summarily hold **Strings**
  - or images
  - or ... anything!
- The trick is getting them back out:
  - When given an **Object**, you have to downcast it

```java
class DumbSet {
    public void insert(Object o) {..}
    public bool member(Object o) {..}
    public Object any() {..}
}

class MyProgram {
    public static void main(String[] args) {
        DumbSet set = new DumbSet();
        String s1 = "foo";
        String s2 = "bar";
        set.insert(s1);
        set.insert(s2);
        System.out.println(s1+"in set?"+set.member(s1));
        String s = (String)set.any(); // downcast
        System.out.println("got " + s);
    }
}
```

**Example**

```java
class DumbSet {
    public void insert(Object o) {...}
    public bool member(Object o) {...}
    public Object any() {...}
}

class MyProgram {
    public static void main(String[] args) {
        DumbSet set = new DumbSet();
        String s1 = "foo";
        String s2 = "bar";
        set.insert(s1);
        set.insert(s2);
        System.out.println(s1 in set?set.member(s1));
        System.out.println(String set.any()); // downcast
        System.out.println("got " + s);
    }
}
```

**Wrapper classes**

- To create **Integer**, **Boolean**, **Double**, …
  - that is a subclass of **Object**
  - useful/required for polymorphic methods
    - **HashTable**, **LinkedList**, ...
  - used in reflection classes
- Include many utility functions
  - e.g., convert to/from **String**
- **Number**: superclass of **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**
  - allows conversion to any other numeric primitive type

**Class Objects**

- For each class, there is an object of type **Class**
- Describes the class as a whole
  - used extensively in Reflection package
- **Class.forName("MyClass")**
  - returns class object for **MyClass**
  - will load **MyClass** if needed
- **Class.forName("MyClass").newInstance()**
  - creates a new instance of **MyClass**
- **MyClass.class** gives the **Class** object for **MyClass**
Array types

- Misfeature: suppose S is a subtype of T
  then S[] is a subtype of T[]

- Object[] is a supertype of all arrays of reference types

Example: Object[]

```java
public class TestArrayTypes {
    public static void reverseArray(Object[] A) {
        for (int i = 0, j = A.length - 1; i < j; i++, j--) {
            Object tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
        }
    }
    public static void main(String[] args) {
        reverseArray(args);
        for (int i = 0; i < A.length; i++)
            System.out.println(args[i]);
    }
}
```

Problem with Subtyping Arrays

```java
public class A { ... }    
public class B extends A { void newMethod(); }
...    
void foo(void) {
    B[] bs = new B[];
    A[] as;
    as = bs;    // Since B[] subtype of A[]
    as[0] = new A();    // (1)
    bs[0].newMethod();    // (2)
}
```

- Program compiles without warning
- Java must generate run-time check at (1) to prevent (2)
  - Type written to array must be subtype of declared type