1 Announcements

- Project 1 (read it and ask questions!)
- Homework 1 (and solution)
- Homework 2 (pair programming)
- Feedback (going up online)
- Reading (how did it go? there will be a catchup day)
- Exam (next week!)
- Autograder (read your class account email!)
- Abstract classes (in lab)

2 Static method calls

- A child class may define a static method with the same number and types and parameters as an inherited method, hiding the inherited method.
- The call $V.f(...)\) is made based on the static type of $V$
- $V$ may be null

Example

class Parent {
    static void foo() {
        System.out.println("parent");
    }
}
class Child extends Parent {
    static void foo() {
        System.out.println("child");
    }
}

static void run() {
    Parent p = new Parent();
    Child c = new Child();
    p.foo(); // prints parent
    c.foo(); // prints child
    p = c;
    p.foo(); // prints parent
}
3 Dynamic method calls

- Non-static methods may be overridden by methods in child classes
- The call \textit{V.f(...)} is made based on the dynamic type of \textit{V}
- If \textit{V} is null, a \texttt{NullPointerException} is thrown

Example 1

```java
class Homer {
    void talk() {
        System.out.println("doh!");
    }
}

class Bart extends Homer {
    void talk() {
        System.out.println("dude");
    }
}

static void run() {
    Homer h = new Homer();
    Bart b = new Bart();
    h.talk(); // prints doh!
    b.talk(); // prints dude
    h = b;
    h.talk(); // prints dude
}
```

Example 2

```java
class Homer {
    void talk() {
        System.out.println(who() + ": doh!");
    }
    void talk2() {
        System.out.print(who() + ", ");
        talk();
    }
    static String who() {
        return "Homer";
    }
}

class Bart extends Homer {
    void talk() {
        System.out.println(who() + ": dude");
    }
    static String who() {
        return "Bart";
    }
}
static void run() {
    Homer h = new Homer();
    Bart b = new Bart();
    h.talk(); // prints Homer: doh!
    b.talk(); // prints Bart: dude
    h = b;
    h.talk(); // prints Bart: dude
    b.talk2(); // prints Homer, Bart: dude
    h.talk2(); // prints Homer, Bart: dude
    (new Homer()).talk2(); // prints Homer, Homer: doh!
}

4 Super

- Allows calls to methods or constructors of parent class
- Just like this except its type is that of the parent class
- Method calls super.f call f based on the type of the parent class

Example

class Parent2 {
    void foo() {
        System.out.print("parent");
    }
}

class Child2 extends Parent2 {
    void foo() {
        super.foo();
        System.out.print("child");
    }
}

static void run() {
    Parent2 p = new Parent2();
    Child2 c = new Child2();
    p.foo(); // prints parent
    c.foo(); // prints parentchild
    p = c;
    p.foo(); // prints parentchild
}

5 Access modifiers

- public indicates that all classes from all packages have access
- protected indicates that all classes within the current package and all child classes have access
• *default* (no access modifier) indicates that all classes within the current package have access

• *private* indicates that only this class (and its inner classes) have access

**Example**

```java
package P;
class A {
    private void private_meth() { }
    void default_meth() { }
    protected protected_meth() { }
    public public_meth() { }
}

class B {
    void run(A x) {
        x.private_meth(); // Compile error
        x.default_meth();
        x.protected_meth();
        x.public_meth();
    }
}

package P2;
class C {
    void run(A x) {
        x.private_meth(); // Compile error
        x.default_meth(); // Compile error
        x.protected_meth(); // Compile error
        x.public_meth();
    }
}

class D extends A {
    void run(A x) {
        x.private_meth(); // Compile error
        x.default_meth(); // Compile error
        x.protected_meth();
        x.public_meth();
    }
}
```

6 **Exceptions**

• Allow for handling of unusual or erroneous conditions

• Can be thrown using *throw E* where *E* is the exception to be thrown

• Can be handled either with a *try { ... } catch (EType v1) { ... }* or by declaring the exception as thrown in the method signature with *throws EType* where *EType* is the type of the exception being handled

• *finally* allows for execution of clean-up code at the end of a try-catch block
- Exceptions are simply classes that inherit from Throwable
- Exceptions that inherit from Error or RuntimeException are considered unchecked, meaning they do not have to be explicitly caught or thrown (consider NullPointerException)