Static typechecking of OO programs

Can use a subclass where a superclass is expected
- how to do typechecking when types don't have to be the same?

Want to make sure that method lookup finds a target method
- how to ensure no `doesNotUnderstand` errors?

Smalltalk doesn't do static typechecking
- \+ flexible
- – can have `doesNotUnderstand` errors at run-time
- – can be harder to understand code, since no interfaces specified

Java and C++ do static typechecking
(Java strongly, C++ weakly)
- \+ ensures no `doesNotUnderstand` errors **before** run-time
- \+ interfaces documented
- – less flexible, more programmer burden

Subtyping

A key notion is **subtyping**:
one type *A* is a subtype of another type *B* if values of type *A* can be used wherever values of type *B* are expected

Each class defines a type

Then, a subclass is a subtype of its superclass(es), since instances of the subclass can be used wherever instances of the superclass are expected

```
POINT3D p3d = ...;
POINT p = p3d;          // OK

POINT q = ...;
POINT3D q3d = q;        // NOT OK: static type error
POINT3D r3d = (POINT3D)q;   // OK statically,
                            // but verified by run-time check

Rectangle r1 = q;       // NOT OK: static type error
Rectangle r2 = (Rectangle)q;  // STILL NOT OK
int i = (int)q;               // STILL NOT OK
```


Subtyping and method calls

Previous slide: rhs of assignment can be subtype of lhs

Also:
- arguments of method calls can be subtypes of the method's declared arguments
- declared result of method call can be subtype of what's expected by context (as with any expression)

```
public interface POINT {
  ...
  public boolean equals(POINT p);
};

POINT3D p = ...;
POINT3D q = ...;
... p.equals(q) ...   // OK, since q's type subtypes
                      // equals's declared argument type
POINT r = p.add(q);   // OK, since add's declared result
                      // type subtypes lhs type
```


Subtyping and method overriding vs. overloading

A rule:
when overriding a method in a subclass, **can't** change the argument types
- if you do, then you're only overloading

Example:

```
POINT add(POINT p)      // inherited from Point
POINT3D add(POINT3D p)  // in CartPoint3D
```

are only **statically overloaded**, not **dynamically overriding**
- a call with a `POINT` argument won't ever invoke the `add(POINT3D)` method, even if the receiver is a `CartPoint3D`

Otherwise, things could go very wrong:

```
POINT3D p = new CartPoint3D(3,4,5);
POINT q = new CartPoint(3,4);
p.add(q);   // invokes add(POINT) inherited from Point;
            // what if it invoked add(POINT3D) instead?
```

Subtyping and method overriding and results

Another rule:

an overriding method **can** change its result type to be a **subtype** of that of the overridden method

Example:

```
public abstract class Point implements POINT {
  ...
  public POINT copy() {
    return new CartPoint(x(), y());
  };
};
public abstract class Point3D
    extends Point implements POINT3D {
  ...
  public POINT3D copy() {
    return new CartPoint3D(x(), y(), z());
  };
};

POINT3D p = new CartPoint3D(3,4,5);
POINT3D q = p.copy();  // OK
POINT r = p;
POINT s = r.copy();     // OK; s will be a CartPoint3D
```


Static typechecking of abstract vs. concrete classes

If a class is abstract, then can't do new on it

```
... new POINT(...) ...       // NOT OK
... new Point(...) ...       // NOT OK
... new CartPoint(...) ...   // OK
```

If a class is concrete, then must ensure that all operations are implemented, either in this class or in a superclass

- must override all interface methods and abstract methods with real implementations

```
public interface POINT3D extends POINT {
  public int z();
  public POINT3D add(POINT3D p);
};
public abstract class Point3D
    extends Point implements POINT3D {
  public abstract int z();
  public POINT3D add(POINT3D p) { ... };
  public String toString() { ... };
};
public class CartPoint3D
    extends CartPoint implements POINT3D {
  // what must be implemented?
};
```