cs2220 Fall 2010
Subtyping and Inheritance Notes (classes 11-13)

Subtyping

Subtyping is a kind of abstraction: it allows us to hide the specific type that is used, mapping many different types to one supertype. *B* is a subtype of *A* means wherever an object of type *A* is expected, we can use an object of type *B* instead.

Inheritance

Inheritance is a way to reuse code. When a subtype inherits an implementation from its supertype, we are reusing the implementation of the supertype to implement a subtype.

Both subtyping and inheritance are general concepts which are implemented by different languages in different ways.

Subtyping and Inheritance in Java

class B extends A means:

- **B** is a subtype of **A**
- **B** inherits the implementation of **A**

class C implements D, E, F means:

- **C** is a subtype of **D**, **E**, and **F**

Should a type be permitted to have more than one subtype?

Should a type be permitted to have more than one supertype? (Why would this be beneficial? Why might it cause problems?)

Should a type be permitted to inherit from more than one superclass? (Why would this be beneficial? Why might it cause problems?)

Apparent and Actual Types

Apparent types are associated with declarations: they never change. The compiler does type checking using apparent type.

Actual types are associated with object: they are always a subtype of the apparent type. The Java Virtual Machine does method dispatch using actual type.

Can the apparent type of an array element ever change?

How can you change the actual type of a variable?

Substitution Principle

B is a subtype of *A* means wherever an object of type *A* is expected, we can use an object of type *B* instead.

For a function *f* (*A*), if *f* satisfies its specification when passed an object whose actual type is type *A*, *f* also satisfies its specification when passed an object whose actual type is *B*.

Signature Rule

- Subtype must implement all of the supertype methods
- Argument types must not be more restrictive (*contravariant*)
- Result type must be at least as restrictive (*covariant*)
- Subtype method must not throw exceptions that are not subtypes of exceptions thrown by supertype (*covariant?*)

Java's rules are different from the signature rule (and the Java 1.5 rule is different from the pre-Java 1.5 rule described in the textbook). The main reason for this is because of the confusion caused by overloading. Argument types must be the same (*novariant*) in overriding subtype methods. Return types may be more restrictive (*covariant*) since Java 1.5, but before Java 1.5 the return types needed to match exactly (*novariant*).

Methods Rule

- Precondition of the subtype method must be weaker than the precondition of the supertype method: m_A.pre *implies* m_B.pre
- Postcondition of the subtype method must be stronger than the postcondition of the supertype method: m_B.post *implies* m_A.post.

If *A implies B*, which is *stronger*?

Remember the logical table for implies:

A => B	**A is true**	**A is False**
B is true		
B is false		

Properties Rule

- Subtype must preserve all properties in the supertype's overview specification.

```
class A {
  public R_A m (P_A p)
        // REQUIRES: pre_A
        // MODIFIES: modifies_A
        // EFFECTS: effects_A
  { … }
}
class B extends A {
  public R_B m (P_B a)
        // REQUIRES: pre_B
        // MODIFIES: modifies_B
        // EFFECTS: effects_B
  { … }
}
```

According to the substitution principle, what are the logical relationships required between:

P_A and *P_B*:

pre_A and *pre_B*:

R_A and *R_B*:

modifies_A and *modifies_B*:

effects_A and *effects_B*:

Subtyping Example

```
MysteryType1 mt1;
MysteryType2 mt2;
MysteryType3 mt3;
… (anything could be here)
mt1 = mt2.m (mt3);
```

If the Java compiler is happy with this code, which of these are guaranteed to be true:

a. The apparent type of mt2 is MysteryType2
b. At the last statement, the actual type of mt2 is MysteryType2
c. MysteryType2 has a method named m
d. The MysteryType2.m method takes a parameter of type MysteryType3
e. The MysteryType2.m method returns a subtype of MysteryType1
f. After the last statement, the actual type of mt1 is MysteryType1