CSE 341 - Programming Languages - Autumn 2012
More on Subtypes and Overloading in Java

Overloading Method Names

A class can have two declarations for a method `test`, one that takes an Object as a parameter and one that takes a String. Here `test` is *overloaded*.

Example:

```
public class Squid {

    public void test(Object x) {
        System.out.println("in the test method that takes an object");
    }

    public void test(String x) {
        System.out.println("in the test method that takes a string");
    }

    public static void main(String[] args) {
        Squid s = new Squid();
        String x = "hi there";
        Object y = x;
        s.test(x);
        s.test(y);
        // note that here the upcast has an effect!!
        s.test( (Object) x);
    }
}
```

Output:

```
in the test method that takes a string
in the test method that takes an object
in the test method that takes an object
```

Note that Java determined which method to call based on the statically determined type of the argument, not its actual type at run time.

However, Java doesn't allow overloading based just on the return type -- so adding these two methods to Squid gives a compile error:

```
    public Object r() {
        return "hi there";
    }

    public String r() {
        return "hi there";
    }
```

Allowing overloading based just on the return type would result in potentially ambiguous expressions. How should `s.test(s.r())` be evaluated?

Java methods can be covariant in the return type

Here's an example:

```
interface Octopus {
    Object test();
}

public class BabyOctopus implements Octopus {
    public String test() {
        return "feed me!";
    }

    public static void main(String[] args) {
        BabyOctopus b = new BabyOctopus();
        Octopus c = b;
        String s = b.test();
        System.out.println(s);
        // this gives a compile error:
        // s = c.test();
        //
        // but this works:
        // s = (String) c.test();
    }

}
```

Rules for the types of method arguments

Suppose that Java didn't overload method names. Would it then be sound to allow method typing to be covariant in the argument types also?

No! This would lead to the same type errors that we saw with the covariant typing rule for Java arrays.

Surprising fact: it *is* sound to allow method typing to be contravariant in the argument types. Example:

```
interface Octopus {
    public void munch(String s);
}

public class BabyOctopus implements Octopus {
    public void munch(Object s) {
        System.out.println("munched an object");
    }
}
```

This gives an error in Java (since Java doesn't use contravariant typing) but would be sound! Anywhere we expected an Octopus we could use a BabyOctopus.