(These notes are very rough, and differ somewhat from what I presented in class; I am posting them here to supplemental your own notes.)

1 Type Checking Java

Before the midterm, we started looking at generics, that is, a way to parameterize interfaces and classes by types.

Last lecture, I would have presented the Java Collections framework, which are a bunch of classes and interfaces distributed in Java that implement sets, and lists, using generic interfaces. The reading is on the web page, and I will ask you to read it carefully, because it essentially counts as a lecture.

Okay, so we have generics. Now the type system has become a bit more complicated. Let's revisit the type system and look carefully at what it guarantees, and how Java establishes that guarantees.

Type checking happens before execution. Type checking can only uses type information that you have declared in your program, and not the possible type that an object has at runtime.

To type check, Java looks at every statement in the code, and make sure that it uses values correctly, according to the declared types. These checks are done using a set of rules, which are called type checking rules. A rule takes the form "if A is true, then B is true"; to simplify things (it doesn't look like a simplification, but it turns out it does), we will write such a rule as:

```
A is true
---------
B is true
```

During type checking, we carry around an environment (called E) that records the the type of the various symbols (variables, class names) that are currently in scope. The initial environment Einit holds the type of all the classes defined in the Java library.

Here are the rules. Let's start at the high-level, type checking a program. A program type checks in environment E (written E |- P ok) if all the classes in the program type check. (Recall that a program is just a sequence of classes.) A class type C checks in environment E (written E |- C ok) if all the fields and methods it contains type check. Thus, to type check a program, we type check every class in the program, assuming that all the classes in the program are added to the environment. (There is a circularity here, but it turns out not to be a problem.)

To type check a class, we need to type checvk all the methods in the class, as well as all the fields. Let's focus on the methods for now. A method type checks (written E |- T name (T1 n1, ..., Tk nk) { ... } ok) if all the statements and declarations it contains type check.

A simple form (without declaration) might be:

```
  E+{n1:T1,...,nk:Tk} |- s1 ok
  ...
  E+{n1:T1,...,nk:Tk} |- sn ok
  ------------------------------------------------------
  E |- T name (T1 n1, ..., Tk nk) { s1; ...; sn; } ok
```

(Declarations just add new bindings to the environment)

Rules for type checking statements (E |- s ok) and expressions that evaluate to a value (E |- ex : T)
Sample statements: assignments (var = ex;) void expressions (ex;), and, say, conditionals (if (e) s else s;)

```
   E |- var : T            E |- ex : T
  ------------------------------------------
      E |- var = ex      ok

   E |- ex : void
   ---------------------
    E |- ex    ok

   E |- ex : boolean    E |- s1 ok   E |- s2 ok
   ------------------------------------------------------------
     E |- if (ex) s1 else s2
```

How about expressions?

Constants are easy:

```
   -----------------
      E |- 1 : int        (and so on for all integer constants)
```

The main rule is type application:

```
  E |- obj : Class {...T name (T1,...,Tn)...}
    E |- e1 : T1   ...   E |- en : Tn
  ------------------------------------------------
  E  |-   obj.name (e1,...,en) : T
```

This says that the type system derives that obj.name (e1,...,en) has type T when e1, ..., en have type T1,..., Tn respectively, and obj has a method "name" of the right type.

So, when type checking, say, System.out.println (x.toString()), the environment has name Foo in it (with type indicating it is a class with methods void sample(Object) and void test()), and also name System which is a class with field out holding an object with a method println expecting a String, and also x with type Object (according to the declaration of the parameters).

Now, x has type Object, so it has a method called toString(), so x.toString() is okay, and the result is a string, so that the call to println is also okay, and returns no value (type void). So the statement type checks.

So in particular, instantiating the above rule to the case above.

```
   E |- System.out : PrintStream {...void println (String)...}
    E |- x.toString() : String
   ------------------------------------------------------------
     E |- System.out.println (x.toString()) : void
```

The above is fine, but we need to establish that x.toString() has type String. Well, this is again established using the above rule:

```
   E |- x : Object (by looking in E) {...String toString()...}
   ------------------------------------------------------------
     E |- x.toString () : String
```

Actually, the real rules are slightly more general:

```
  E |- obj : Class {...T name (T1,...,Tn)...}
    E |- e1 : U1   ...   E |- en : Un
      E |- U1 <= T1  ...   E |- Un <= Tn
  ------------------------------------------------
  E  |-   obj.name (e1,...,en) : T

   E |- var : T        E |- ex : U     E |- U <= T
  ------------------------------------------------
     E |- var = ex      ok
```

where E |- U <= T means: if U is a subtype of T in environment E
(which defines, for instance, what subclasses are subclasses of what
classes)

```
   -------------
   E |- U <= T    if E says that U <= T (e.g. U = class that implements T)

   -------------
   E |- T <= T


   E |- U <= T
   -----------------------
   E |- U[] <= T[]

   E |- U <= T   E |- T <= S
   ---------------------------------
     E |- U <= S
```

This more general rule is used to type check:

```
       this.sample (i);
```

in the second method.

```
E |- this : Foo {...void sample (Object)...}
 E |- i : Integer   (from E)
  E |- Integer <= Object  (from E)
 ------------------------------------------
 E |- this.sample(i) : void
```

So there is a well defined notion of what it means for a program to type check.

What does type checking guarantees?

It says that some bad things cannot happen at runtime.

At runtime, every object created has an associated run-time class (the class it was created as). Operationally, at run-time, when an object is passed to a method that may expect a superclass, the extra methods are not deleted, they are still there, although the method may not be able to access them (because type checking ensured that the system cannot look at them).

The main runtime guarantees: if Einit |- P ok, then during execution of the program, whenever the system attempts to invoke method M on some object O, then object O implements method M.

(In older OO languages that did not have static type checking, we could have an exception "Method undefined". We never get this for Java. The above is often called soundness.)