CSE 11: Lecture 18

- Compile-time and run-time type checking
- Type checking and dynamic method binding
- Dynamic binding: calling object vs. method arguments
- Quick review of CSE 11 topics

Final Exam will be...

Friday Dec 12 8:00am-10:30am

Location: here

Closed-book, closed-notes, no calculators. Bring picture ID!

Coverage: Everything...

A practice final is available online (PDF format)

Lecture notes are available online

Final review session will be: today

The importance of types and type-checking

- Recall that a variable is just a collection of bits (1's and 0's) in the computer's memory
- How that pattern of bits is interpreted as a meaningful value and what operations are permitted on it in a high-level language program depends on the *data type* of the variable
- So it is important that the operations performed on a variable are appropriate to its type! Otherwise you have 'garbage in, garbage out'.
- Java does *type checking* to try to ensure that inappropriate things don't happen
- Java does two kinds of type checking:
  - Compile-time (also called "static") type checking
  - Run-time (also called "dynamic") type checking

Compile-time type checking

- Java is a “strongly typed” language
- This means: The compiler can determine the datatype of every expression in your program at compile time
- If your program code tries to do something with an expression that is inappropriate to the type of the expression, it is a compile-time error
- This is good: The compiler is trying to prevent ‘garbage in, garbage out’ bugs in your program
- But to prevent these compile-time type errors, or to fix ones the compiler has found in your code, you have to clearly understand Java’s type checking rules

Determining the type of an expression at compile time

- In a strongly typed language, the compiler can determine the type of every expression. How does it do that, in Java?
- It depends on what kind of expression it is:

to tell the type of a...	the compiler looks at...
identifier	the type specified in the identifier's declaration
operator application	the type rules for the operator, given the type(s) of the operator's argument(s)
method call	the return type specified in the method header
cast expression	the typename in the cast
result of "new"	the name of the constructor that is called
literal constant	the syntax of the literal constant
indexed array expression	the type specified in the array declaration

Type rules enforced at compile time

- The type of every expression can be determined by the compiler, but some types are not allowed in some contexts, to try to prevent potentially meaningless operations... These are the rules the compiler will enforce:

- Arguments to numerical, comparison, and boolean operators: See Lecture 3

- Arguments to the assignment operator (see Lecture 5):
  - You can assign an expression of a certain type to a variable of the same type.
  - You can assign an expression of a numerical type to a variable of any "wider" type.
  - You can assign an expression of boolean type to a variable of boolean type.
  - You can assign an expression of reference type to a variable of any ancestor type.
  - Any other assignment is a compile-time error.

- Passing arguments to a method: same rules as for assignment; substitute "pass" for "assign", and "formal parameter" for "variable"

Type rules enforced at compile time, cont'd

- Casts:
  - You can cast an expression of a certain type to the same type.
  - You can cast a numerical type expression to any other numerical type.
  - You can cast a reference type expression to any ancestor type ("upcast") or descendant type ("downcast").
  - Any other cast is a compile-time error.

- Member access ("dot" operator), expression of the form `expr1 . expr2` :
  - The method or variable named by `expr2` must exist (and be visible) in the reference type of `expr1`, either by being defined there, or by being inherited from an ancestor type
  - If `expr2` is a method call, the types of the actual argument expressions have to match the types of corresponding formal parameters
  - Any other member access is a compile-time error.

Run-time errors

- The strong typing of Java permits many type checks to be done at compile-time, and this prevents many bugs from making their way into running programs
- However, compile-time type checking does not prevent all bugs!
- There are still run-time errors and logic errors that can occur
- Some run-time errors that are detected as a result of run-time type checking are:
  - Class cast exceptions, from failed downcasts
  - Array store exceptions, from failed array element assignments
- And there are many run-time errors that have nothing to do with run-time type checking, for example:
  - Array index out of bounds
  - Trying to access a member of a null pointer
  - I/O exceptions
  - Running out of memory, stack overflows, etc.

Type rules enforced at run-time

- Class cast exceptions, from failed downcasts

```
Object o = new Object();
String s = (String) o;  // Class cast exception: o does not
                        // point to a String
```

- Array store exceptions, from failed array element assignments

```
Object[] o = new String[1];
o[0] = new Object();   // Array store exception: o points to
                       // an array whose elements are
                       // pointer-to-String
```

Compile-time type checking and dynamic method binding

- Non-final public instance methods are dynamically bound in Java: The version of the method called depends on the type of the object pointed to, not the type of the pointer
- Dynamic method binding happens at run-time...
- But to get to run time, your code must first compile!
- ... and to compile, the method call must pass compile-time type checking

Type checking and dynamic binding, example:

```
public class BB {
   public BB() { x = 3; }
   public double foo() {return x + 1.0;}
   private double x;
}
-----------------------------------------------
public class DD extends BB {
   public DD() { y = 4; }
   public double foo() {return y + 2.0;  }
   public double foo(int d) {return y + d; }
   public double bar() {return foo(); }
   private double y;
}
-----------------------------------------------
BB b = new DD();
double s = b.foo();   // OK: foo() is in BB; dyn. binding happens
double t = b.bar();  // COMPILE-TIME ERROR:  bar() is not in BB
double u = b.foo(7); // COMPILE-TIME ERROR: foo(int) is not in BB
DD c = new DD();
double v = c.foo(7);  // OK: foo(int) is in DD
```

The “calling object” vs. method arguments

- In object-oriented programming, you design, define, and call instance methods of objects
- When an instance method is called, there is an object whose instance method you are calling
  - This object is termed the “calling object”, or the “receiver”, or the “this” object, or the “self” object, etc.
- In some ways you can think of the calling object as just another argument to the method
- For example this instance method in the Object class

```
public boolean equals(Object other) {
   return this == other;
}
```

  is very similar to a method you might write as

```
public static boolean equals(Object this, Object other) {
   return this == other;
}
```

- The big difference is: dynamic binding happens for the calling object, but
- *not for any argument objects*!

The “calling object” vs. method arguments

- Dynamic binding happens for the calling object, not for any argument objects!

```
public class A {
  public void f(A x) { System.out.println("A A"); }
  public void f(B x) { System.out.println("A B"); }
}
public class B extends A {
  public void f(A x) { System.out.println("B A"); }
  public void f(B x) { System.out.println("B B"); }
}
...
public static void main(String args[]) {
   A a1 = new A();  A a2 = new B();  B b = new B();
   a1.f(a1);      // A A
   a1.f(a2);      // A A
   a1.f(b);       // A B
   a2.f(a1);      // B A
   a2.f(a2);      // B A
   a2.f(b);       // B B
```

Final exam topics

- Any topic in the textbook that we have mentioned in lecture
- Anything discussed in lecture or used on a programming assignment
- Main topics:
  Unix filesystem structure; Unix commands
  Aspects of algorithm design, implementation, and testing
  Abstract data types: interface and implementation
  Primitive Java types; arithmetic and boolean operators
  Variables: declaration, initialization, assignment, and use
  Control constructs: if, if-else, while, do-while, for, switch
  Standard library classes, and user-defined class types
  Static and instance methods and variables; constructors
  Primitive and class type method arguments
  Exception handling
  Terminal and file I/O
  Class derivation and Inheritance
  Public, private, protected, and “package” visibility
  Dynamic method binding
  Method overloading and overriding
  Event-driven programming and JFC/Swing; applets
  Arrays
  Packages

What's left to learn about programming and Java?

- ✔ You've learned a lot this quarter...
  - ✗ With that knowledge comes the power to do things you couldn't do before.
  - ✗ Use your knowledge wisely!

- ✔ What's left to learn? Well, quite a bit, actually.

  For examples, here's a partial list of topics covered in the next course, CSE 12:

  - ✗ Data structures: stacks, queues, linked lists, sets, trees, heaps, hash tables
  - ✗ Strategies for generic ADT design
  - ✗ Bitwise operators
  - ✗ Recursion
  - ✗ Pointer arithmetic and low-level data representations
  - ✗ Asymptotic analysis of algorithm cost
  - ✗ Etc.!