Software Engineering 301:
Software Analysis and Design

Review of object-orientation

Agenda

- Objects, classes, types, fields, variables, state
- Methods, operations, constructors
- Subtyping/supertyping, inheritance, polymorphism
- Nested types
- Generics
- Reflection
- Exceptions

not actually OO, but useful and common

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

Here is a Java source file that we'll use as an example.

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
  public ScriptProcessor(String path, IVendingMachineFactory factory,
          boolean debug) throws FileNotFoundException, ParseException {
    Parser p = new Parser(new FileReader(path));
    p.register(factory);
    p.setDebug(debug);
    p.process(path);
  }
}
```

What is this? When does it execute?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

How many objects are explicitly created in this piece of code?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

When are those objects created?

When are those objects destroyed?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p;
      while(true) {
         p = new Parser(new FileReader(path));
         p.register(factory);
         p.setDebug(debug);
         p.process(path);
      }
   }
}
```

How many objects are explicitly created in this piece of code?

When are those objects created? Destroyed?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

How many objects have to exist before this code can execute?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

Where are the objects?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

How many classes or interfaces are used or defined in this code?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

How many types are used or defined in this code?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

How many types are used or defined in this code?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
         boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

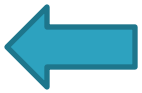
What is "p"? "path"? "factory"? "debug"?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p;
      while(true) {
         p = new Parser(new FileReader(path));
         p.register(factory);
         p.setDebug(debug);
         p.process(path);
      }
   }
}
```



Notice that "p" is set repeatedly to a different object.

Equivalent: "p" is reassigned in each iteration.

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
         boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

When is "p" used? When is the object in "p" used?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
         boolean debug) throws FileNotFoundException, ParseException {
      Parser p = null;
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

When is "p" used? When is the object in "p" used?

Objects

- Objects are conceptual entities, a metaphor from the real world
  - In the real world, an object is something you can point to, that has an individual identity
- This concept breaks down if you push too hard
  - Is water an object?
  - Objects can be subdivided, eventually to molecules, atoms, etc.

Objects

- The real world is largely continuous, but objects are discrete
  - We can often conveniently ignore the difference
- Choosing the objects to use in our software is an important choice
  - Design will revolve around this
- The source code does not mention specific objects
  - “new”, when executed, indicates that an object should be created

Classes

- We can group objects into categories (i.e., **classify** the objects) in which all the objects there have at least common kinds of properties
  - The specific values of those properties may differ
  - If all properties of two objects are identical, are there really two objects?
    - Consider equals() vs == in Java

Classes

- Equivalently, we can define a class as consisting of all objects that possess certain kinds of properties
  - If class Student has property StudentID, only objects with this property can be in this class
  - If an object has property StudentID, this may not suffice for it to be in the class Student

Objects vs classes

- An object can be an **instance** of a class
  - An object (in principle) can be an instance of several classes at once
  - Each of you is an instance of Human, Student, SENG 301 Student
    - You are each an instance of Child; some of you may be an instance of Parent
    - There are lots of other classifications that apply to some of you
- A class is **instantiated** to create an object

Types in Java

- Primitive: int, char, boolean, etc.
- Reference:
  - Arrays: int[], double[][], byte[][][], etc.
  - Classes
    - enumerations (enums)
  - Interfaces  *more later*
    - annotation types
- *We will examine the differences later …*

Objects and classes in Java

- Java is actually a class-oriented language
  - We write classes, and objects are created by selecting the class of interest
- Java does not permit the class of an object to change
- Java does not permit an object to be a direct instance of more than one class

Other types in Java

- Arrays
  - `int[] arr = new int[2]`
  - `int[] myField` is equivalent to `int myField[]`
- Enumerations
  - Typesafe alternative to multiple constants
- Annotation types
  - permit special properties to be enforced on entities
  - e.g., `@SuppressWarnings("unused")`

References vs. primitive types

- Reference types: things that can be pointed to
  - classes, interfaces, enumerations, annotation types, arrays
- Primitive types
  - no references or pointers to these (unlike C/C++)

Fields

- Class properties are realized as **fields** in Java
  - Called **member variables** in C++
- Each instance of a class will typically have its own copy of the fields defined by its class
  - This is actually only true for **instance fields**
  - Fields can also be shared across all instances (**class fields** or **static fields**)
- Driver declares no fields, but this does not mean it *has* no fields!

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   private Object myFakeField = null;

   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

Other variables in Java

- Formal parameters to methods/constructors are variables with scope only within that method/constructor
- Local variables can also be defined, which have scope only from the declaration until the end of the enclosing scope (block, for-loop, etc.)
- All variables have a type in Java

State

- The value of all fields in an object constitutes its current **state**
  - Sometimes, only certain fields are relevant to us
- A program's state includes the state of all its objects plus the state of its variables
- For some programs, *external data* (in files and databases) is also relevant to its state
- State only exists at specific moments of *runtime*

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

What is the state of this code?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
         boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}
```

What is the state of this code, when execution reaches the arrow?

```
package org.lsmr.vending.frontend1;

import java.io.FileNotFoundException;
import java.io.FileReader;

import org.lsmr.vending.frontend1.parser.ParseException;
import org.lsmr.vending.frontend1.parser.Parser;

public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p;
      while(true) {
         p = new Parser(new FileReader(path));
         p.register(factory);
         p.setDebug(debug);
         p.process(path);
      }
   }
}
```

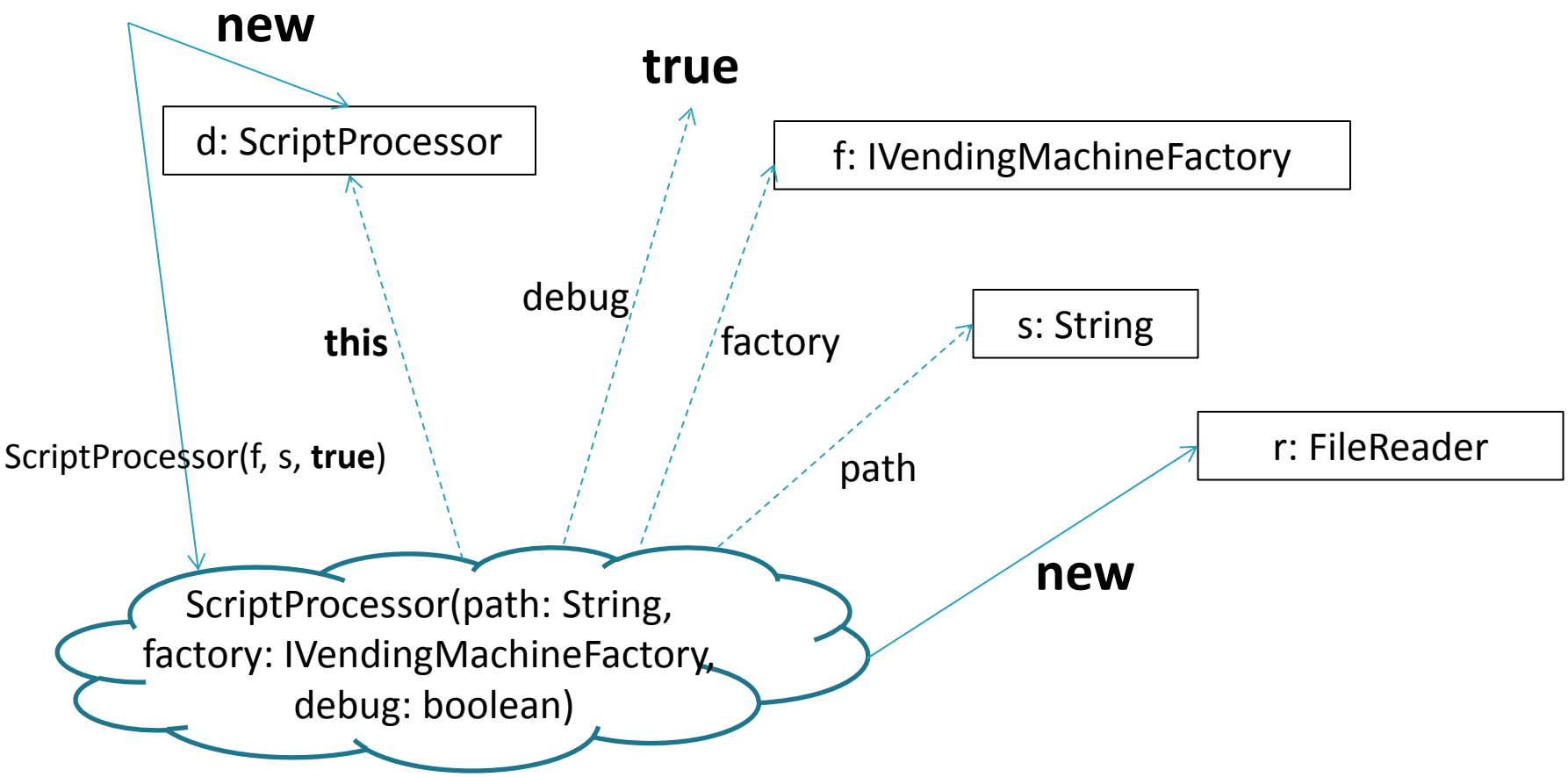
What is the state of this code, when execution reaches the arrow?

```
public class ScriptProcessor {
   public ScriptProcessor(String path, IVendingMachineFactory factory,
            boolean debug) throws FileNotFoundException, ParseException {
      Parser p = new Parser(new FileReader(path));
      p.register(factory);
      p.setDebug(debug);
      p.process(path);
   }
}

public class Parser {
   private IVendingMachineFactory vm = null;
   public void register(IVendingMachineFactory vm) {
      this.vm = vm;
   } …
```

What happens when the constructor executes?

What actually happens



NOTE: This is not UML!

Agenda

- ~~Objects, classes, types, fields, variables, state~~
- Methods, operations, constructors
- Subtyping/supertyping, inheritance, polymorphism
- Nested types
- Generics
- Reflection
- Exceptions

Operations and methods

- Allow computations to be performed or data to be accessed/changed
- An operation is a concept
- A method is the implementation of an operation
- A given operation can have multiple methods that implement it
  - In Java, this only happens in different classes

Operations and methods

- Have properties:
  - Name
  - Formal parameters
  - Result
  - Declared exceptions
  - Visibility and other modifiers
- A signature is the means by which an operation can be identified and differentiated
  - In Java, two operations with identical name and list of parameters are indistinguishable

Operations/methods

- An operation is **called** or **invoked**, which causes a method to be **executed**
  - More sloppily, we often say that a method is called or invoked
- When an object A invokes an operation on another object B, we can also talk equivalently about A **sending a message** to B
  - More precisely: a method (or constructor) that has been executed on A can invoke an operation on B

Formal parameters versus arguments

- An operation/method declares a set of formal parameters
- When an operation is called, a set of arguments is sent to the method
  - Each of these is bound to the relevant formal parameter
  - Equivalently, each formal parameter is **initialized** with the passed arguments

Agenda

- ~~Objects, classes, types, fields, variables, state~~
- ~~Methods, operations, constructors~~
- Subtyping/supertyping, inheritance, polymorphism
- Nested types
- Generics
- Reflection
- Exceptions

```
class A {
   void m() {
      System.out.print("A");
   }
}

class SubA extends A {
   void m() {
      System.out.print("SubA");
   }
}
```

```
class Client {
   public static void
   main(String[] args) {
      A myA = new A();
      Client myClient = new Client();
      myClient.doit(myA);
   }

   void doit(A someA) {
      someA.m();
   }
}
```

What gets printed?

```
class A {
   void m() {
      System.out.print("A");
   }
}

class SubA extends A {
   void m() {
      System.out.print("SubA");
   }
}
```

```
class Client {
   public static void
   main(String[] args) {
      A myA = new SubA();
      Client myClient = new Client();
      myClient.doit(myA);
   }

   void doit(A someA) {
      someA.m();
   }
}
```

What gets printed?

```
interface IEmmable {
   void m();
}

class A implements IEmmable {
   void m() {
      System.out.print("A");
   }
}

class B implements IEmmable {
   void m() {
      System.out.print("B");
   }
}
```

```
class Client {
   public static void
   main(String[] args) {
      IEmmable myA = new SubA();
      Client myClient = new Client();
      myClient.doit(myA);
   }

   void doit(IEmmable emmable) {
      emmable.m();
   }
}
```

What gets printed?

```
class A {
   void m() {
      System.out.print("A");
   }
}

class SubA extends A {
   void m() {
      System.out.print("SubA");
   }
}
```

```
class Client {
   void doit(A someA) {
      someA.m();
   }
}
```

What gets printed when Client.doit() is invoked?

```
interface IEmmable {
   void m();
}

class A implements IEmmable {
   void m() {
      System.out.print("A");
   }
}

class B implements IEmmable {
   void m() {
      System.out.print("B");
   }
}
```

```
class Client {
   void doit(IEmmable emmable) {
      emmable.m();
   }
}
```

What gets printed when Client.doit() is invoked?

Class hierarchies

- Classes are essentially *sets* of objects
- Subsets and supersets have analogous ideas with respect to classes
  - A **superclass** is a more general class, in which fewer properties might be defined (but not more)
  - A **subclass** is a more specific class, in which more properties might be defined (but not fewer)
  - In Java: `class A extends B` means A is a subclass of B (and that B is a superclass of A)

Class hierarchies

- Two sets may overlap, and conceptually, two classes may overlap
  - This means that objects in the intersection are instances of two different classes simultaneously
  -  This is called **multiple inheritance**
    - Multiple inheritance is problematic for correct language design, so most languages either forbid it or restrict how it works

Interface types in Java

- To permit a form of multiple inheritance, Java provides **interface types** (or simply, **interfaces**)
  - A class realizes ("implements") zero or more interfaces
  - Each interface declares the operations that its realizing classes have to implement
    - This is extremely useful in **polymorphism** (later in the lecture)
  - If `class A implements B`, B is an interface and A is a subtype of B
  - If `interface A extends B`, A and B are both interfaces and A is a subtype of B

Supertypes/subtypes

- Only meaningful for reference types
- Supertype is analogous to superclass
- Subtype is analogous to subclass
  - Supertype would include the realized interface types, etc.

Inheritance

- Let SubA be a subclass of A
- SubA will **inherit** all non-private fields and methods of A
- This means that any additional methods of SubA can access all non-private fields and methods of A (but A will not know anything about SubA, usually)

Overloading

- In Java, different operations (and hence methods) can have identical names if they have different formal parameters
- Such methods are said to **overload** each other

Operations and methods

- In Java, methods can be defined as **abstract**
  - Meaning that they provide no implementation
  - This is roughly equivalent to pure virtual functions in C/C++
- This means that the class that declares such an abstract method must also be abstract
  - Abstract classes cannot be directly instantiated
  - “Concrete” is the opposite of “abstract”

Overriding and polymorphism

- Say that SubA inherits a method M from A
- Say also that SubA implements a method M' such that the signatures of M and M' are indistinguishable
- M' is said to **override** M
- Calls to the operation M may get rerouted to execute M' instead
  - This is called **polymorphism**

Implementing polymorphism

- Each class is represented by a table
  - one column is the operation signatures
  - one column is the pointer to the methods for that class
- An instance of class is given a copy of this table
- A operation call on an object A looks for the most appropriate operation in the table, and invokes the method pointed to by A

Agenda

- ~~Objects, classes, types, fields, variables, state~~
- ~~Methods, operations, constructors~~
- ~~Subtyping/supertyping, inheritance, polymorphism~~
- Nested types
- Generics
- Reflection
- Exceptions

Nested types

- Java allows types to be defined inside of other types
  - This allows the **nested type** to access fields and operations of the nesting type
- Nested types can be static or instance-level
  - Meaning they have access to a specific instance of the nesting type
- Usually, the nested types are simple
- We can say that the nesting type **contains** the nested type

Agenda

- ~~Objects, classes, types, fields, variables, state~~
- ~~Methods, operations, constructors~~
- ~~Subtyping/supertyping, inheritance, polymorphism~~
- ~~Nested types~~
- Generics
- Reflection
- Exceptions

Generics

- Imagine you have a collection of strings and another collection of integers
  - Do you need to implement two collection classes?
- To avoid the need for special implementations, **generic types** allow other types to be parameterized
  - The generic type is implemented relative to a type parameter
  - The type parameter has to be bound to an actual type

Generics: example

```
class IntegerArrayList {
   boolean add(Integer e) { … }
   Integer get(int index) { … }
}

class StringArrayList {
   boolean add(String e) { … }
   String get(int index) { … }
}

… + other variations
```

Instead:

```
class ArrayList<E> {
   boolean add(E e) { … }
   E get(int index) { … }
}
```

To use:

```
ArrayList<Integer>
ArrayList<String>
ArrayList<?>
```

Exercise

```
public abstract class AbstractHardware<T extends AbstractHardwareListener> {
   protected Vector<T> listeners = new Vector<T>();

   public final void register(T listener) {
      listeners.add(listener);
   }

   public final void disable() {
      Class<?>[] parameterTypes = new Class<?>[] { AbstractHardware.class };
      Object[] args = new Object[] { this };
      notifyListeners(AbstractHardwareListener.class, "disabled",
         parameterTypes, args);
   }
   ….
}
```

Agenda

- ~~Objects, classes, types, fields, variables, state~~
- ~~Methods, operations, constructors~~
- ~~Subtyping/supertyping, inheritance, polymorphism~~
- ~~Nested types~~
- ~~Generics~~
- Reflection
- Exceptions

Reflection

- Having the program analyze itself and manipulate itself indirectly
- Moves compile-time knowledge about types to the run-time
  - Positive: makes programs more flexible
  - Negative: makes them harder to reason about, harder to write, harder to debug

Reflection: example

```
void m() {
   C someC = new C();
   someC.doit();
}
```

Instead:

More complicated than this because of exception handling

```
void m() {
   Class cls = Class.forName("C");
   Method m = cls.getMethod("doit", new Class[0]);
   m.invoke(new Object[0]);
}
```

```
protected final void notifyListeners(Class<?> listenerClass,
   String eventNotificationMethodName,
   Class<?>[] parameterTypes,
   Object[] args) {
   try {
      Method m =
         listenerClass.getMethod(eventNotificationMethodName,
            parameterTypes);

      for(T listener : listeners)
         m.invoke(listener, args);
   }
   catch(Exception e) {
      throw new SimulationException(e);
   }
}
```

Agenda

- ~~Objects, classes, types, fields, variables, state~~
- ~~Methods, operations, constructors~~
- ~~Subtyping/supertyping, inheritance, polymorphism~~
- ~~Nested types~~
- ~~Generics~~
- ~~Reflection~~
- Exceptions

Exceptions

- Exceptions are a way for a method to signal that something went wrong
- This does not mean it was “unexpected”
- e.g., consider a method that requires an index into an array
  - values that are pass the ends of the array should cause an exception
- Exceptions are **raised** or **thrown**
- Exceptions can be **caught** or **handled** if the calling method knows what to do in such a case

Exceptions in Java

- Two kinds: declared and undeclared
- Undeclared exceptions have `RuntimeException` as an ancestor class
- Declared exceptions must either be handled or declared in a calling method

```
protected final void notifyListeners(Class<?> listenerClass,
   String eventNotificationMethodName,
   Class<?>[] parameterTypes,
   Object[] args) {
   try {
      Method m =
         listenerClass.getMethod(eventNotificationMethodName,
            parameterTypes);

      for(T listener : listeners)
         m.invoke(listener, args);
   }
   catch(Exception e) {  // usually, "Exception" is too general to use here
      throw new SimulationException(e);
   }
}
```

Next time

Structural modelling