CS201: Data Structures and Discrete Mathematics I

Java Topics

Java features

- Secure, platform-independent software execution.

  myProgram.java -> Java compiler -> <myProgram.class>/<bytecode> ->
  interpreter/Java virtual machine(JVM)

- Without pointer problems of C/C++
- A simple built-in memory management/garbage collection.
- Simple constructs for multiprocessing, networking, graphic user interface, etc.

http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html

Objects, Objects, Objects

- In Java, *everything* is an Object.
  - Except primitive types, boolean, char, byte, short, int, long, float, double
- Java source code is based on classes
  - in general: one public class defined in one file.
- Java has a lot of pre-defined classes

Java Program, Compiler and Interpreter

- To begin, you have to create a class!
- You run the class.
  - for this to work, the class must have a method named main() that is declared as

```
public static void main(Strings[] arg)
```

- Compiler: `javac` *`filename`*`.java`
  - Creates *`filename`*`.class`
- Interpreter: `java` *`classname`*
  - You tell the interpreter a class to run

`public static void main`

- `public`: This method can be accessed (called) from outside the class.
- `static`: This method does not require that an object of the class exists. static methods are sort-of like "global" methods, they are always available (you need to use the class name to get at them).
- `void`: no return value.

Language Elements

Types

Literals

Variables

Operators

Control Structures

Exceptions

Arrays

Data Types

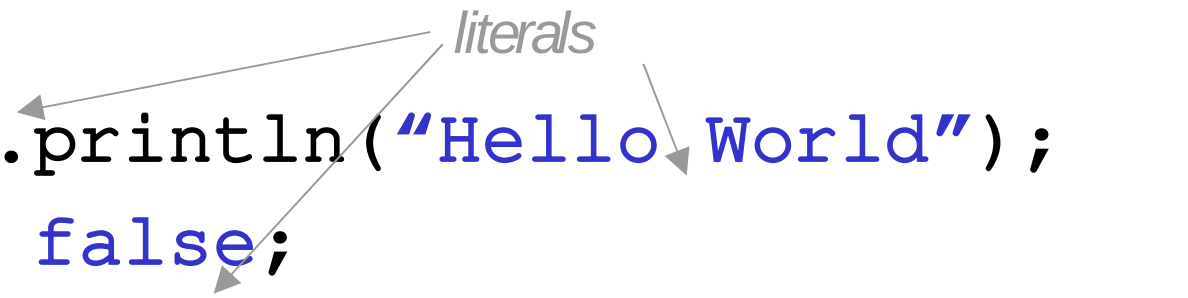
- Everything that has a *value* also has a *type.*
  - anything that can be stored in a variable, passed to a method, returned from a method or operated on by an operator.
- In Java there are two kinds of types:
  - Primitive Types: byte, *char, int, short, long, float, double, boolean*
  - Reference Types
    - *Everything else, objects, arrays, interfaces*

Literals

- Literals are fixed values found in a program.

- Examples:

```
x = y + 3;
System.out.println("Hello World");
finished = false;
```



Variables

- Storage location (chunk of memory) and an associated type.
  - type is either a primitive type or a reference type.
- For primitive type variables, a variable holds the actual value (the memory is used to store the value).
- For reference type variables, a variable holds a reference to an object.

Operators

- arithmetic/logical
- assignment
- boolean/relational (comparison)
- string

Control Structures

```
if
if/else
do
while
for
switch
```

Syntax is very similar (in most cases identical) to C/C++

Exceptions

- The Java language provides support for handling errors (or any kind of unusual condition).
- If you want to use any Java libraries you need to understand exceptions (and you can't do much of anything in Java unless you use libraries!).
- When an *exception* occurs, control is *thrown* to a special chunk of code that deals with the problem.

Arrays

- Array syntax is very similar to C/C++
- Every array access is checked (at run time) to make sure it is within the bounds of the array.
  - arrays have fixed sizes
- The length of an array is available as a *field* of the array.

Creating an Array

- You use the Java `new` operator:

```
foo = new int [100];
studentNames = new String[20];
```

- You often see both declaration and creation like this:

```
int [] foo = new int [100];
String [] studentNames = new String[20];
```

Object Oriented Programming (OOP) and Java

Structured Programming

- Back in the "old days" we had Structured Programming:
  - data was separate from code.
  - programmer is responsible for organizing everything in to logical units of code/data.
  - no help from the compiler/language for enforcing modularity, …

OOP to the rescue

- Keep data *near* the relevant code.
- Provide a nice packaging mechanism for related code.
- Model the world as objects.
- objects can send "messages" to each other.

An Object

- Collection of:
  - Fields (object state, data members, instance variables, ..)
  - Methods (behaviors, …)
- Each object has it's own memory for maintaining state (the fields).
- All objects of the same type share code.

Modern OOP Benefits

- Code re-use
  - programmer efficiency
- Encapsulation
  - code quality, ease of maintenance
- Inheritance
  - efficiency, extensibility.
- Polymorphism
  - power!

Code Re-Use

- nice packaging makes it easy to document/find appropriate code.
- everyone uses the same basic method of organizing code (object types).
- easy to re-use code instead of writing minor variations of the same code multiple times (inheritance).

Encapsulation

- Information Hiding.
- Don't need to know how some component is implemented to use it.
- Implementation can change without effecting any calling code.

Inheritance

- Take an existing object type (collection of fields and methods) and extend it.
  - create a special version of the code without re-writing any of the existing code (or even explicitly calling it!).
  - End result is a more *specific* object type, called the sub-class / derived class / child class.
  - The original code is called the superclass / parent class / base class.

Inheritance Example

- Employee: name, email, phone
  - FulltimeEmployee: also has salary, office, benefits, …
    - Manager: CompanyCar, can change salaries, rates contracts, offices, etc.
  - Contractor: HourlyRate, ContractDuration, …

- A *manager* a special kind of FullTimeEmployee, which is a special kind of Employee.

Polymorphism

- Create code that deals with general object types, without the need to know what specific type each object is.
- Generate a list of employee names:
  - all objects derived from Employee have a name field!
  - no need to treat managers differently from anyone else.

Method Polymorphism

- The real power comes with method overloading
- For example:
  - shape object types used by a drawing program.
  - we want to be able to handle any kind of shape someone wants to code (in the future).
  - we want to be able to write code now that can deal with shape objects (without knowing what they are!).

Shapes

- Shape:
  - color, layer fields
  - draw() draw itself on the screen
  - calcArea() calculates it's own area.
  - serialize()generate a string that can be saved and later used to re-generate the object.

Kinds of Shapes

- Rectangle
- Triangle
- Circle

Each could be a kind of shape (could be specializations of the shape class).

Each knows how to draw itself, etc.

Could write code to have all shapes draw themselves,

Java class definition

```
class classname {
    field declarations
    { initialization code }
    Constructors
    Methods
}
```

Creating an Object

- Make an instance of a a class :

```
classname varname = new classname();
```

- In general, an object must be created before any methods can be called.
  - the exceptions are *static* methods.
- An object is a chunk of memory:
  - holds field values
  - holds an associated object type
- All objects of the same type share code
  - they all have same object type, but can have different field values.

Constructors

- You can create multiple constructors, each must accept different parameters.
- One constructor can call another.
- You use "this", not the classname:

```
class Foo {
   int i;
   Foo() {
      this(0);
   }
   Foo( int x ) {
      i = x;
   }
```

`Abstract` Class modifier

- Abstract modifier means that the class can be used as a superclass only.
  - no objects of this class can be created.

- Used in inheritance hierarchies…(more on this later).

Field Modifiers

- Fields (data members) can be any primitive or reference type.
  - As always, declaring a field of a reference type does not create an object!
- Modifiers:
  - `public private protected static final`
  - there are a few others…

`public/private/protected` Fields

- `public`: any method (in any class) can access the field.
- `protected`: any method in the same *package* can access the field, or any derived class.
- `private`: only methods in the class can access the field.
- default is that only methods in the same package can access the field.

`static` fields

- Fields declared `static` are called *class fields* (*class variables).*
  - others are called *instance fields*.

- There is only one copy of a static field, no matter how many objects are created.

`final` fields

- The keyword `final` means: once the value is set, it can never be changed.

- Typically used for constants, e.g., pi.

```
static final int  BUFSIZE=100;
   final double PI=3.14159;
```

Method modifiers

- **private/protected/public**:
  - same idea as with fields.
- **abstract**: no implementation given, must be supplied by subclass.
  - the class itself must also be declared **abstract**
- **static**: the method is a *class method*, it doesn't depend on any instance fields or methods, and can be called without first creating an object.
- **final**: the method cannot be changed by a subclass (no alternative implementation can be provided by a subclass).

Method Overloading

- You can overload methods:
  - same name, different parameters.
  - you can't *just* change return type, the parameters need to be different.
- Method overloading is resolved at compile time.

```
int CounterValue() {
 return counter;
}
double CounterValue() {
 return (double) counter;
}
```

Inheritance

- One object type is defined as being a special version of some other object type (using **extends**)
  - a *specialization*.
- The more general class is called:
  - base class, super class, parent class.
- The more specific class is called:
  - derived class, subclass, child class.

Java Inheritance

- Two kinds:
  - implementation: the code that defines methods.
    - Derived class inherits the implementations of all methods from base class.
    - can replace some with alternatives.
    - new methods in derived class can access all non-private base class fields and methods.
  - interface: the method prototypes only.

Single inheritance only (implementation inheritance).

- You can't *extend* more than one class!
  - the derived class can't have more than one base class.
- You can do multiple inheritance with *interface inheritance.*

Interfaces

- An interface is a definition of method prototypes and possibly some constants (static final fields).
- An interface does not include the implementation of any methods, it just defines a set of methods that could be implemented.

```
public interface sellable {
    public String description();
    public int listprice()
    public int lowestprice();
}
```

interface implementation

- A class can **implement** an interface, this means that it provides implementations for all the methods in the interface.
- Java classes can implement any number of interfaces (multiple interface inheritance).

```
public class photograph implement Sellable {
    private String descript;
    private int price
    …
}
```

Exceptions

Exceptions, throw and catch

- Exceptions are unexpected events that occur during the execution of a program.
- In Java, exceptions are objects that are "***throw***" by code that encounters some sort of unexpected conditions.
- A thrown exception is ***caught*** by other code that handles the exception. Otherwise, the program terminates.

Some issues

- What to do when you catch an exception?
- How and when to generate exceptions.
- `RunTime` exceptions.
- Custom Exception types.
- Using `finally.`

How/when do you *generate* exceptions?

- Use throw:

```
If (insertIndex > size()) {
   throw new BoundaryViolationException("no
        element at index" + insertIndex);
```

- You can use throw anywhere.
  - you detect some error that means the following code should not be executed.

Claiming exceptions

- When a method is declared, we can specify the exceptions it might throw.
- By doing this, you prepare others to handle all the exceptions.

```
public void goShopping() throws
    ShoppingListTooSmallException,
    OutOfMoneyException {
        // method body
}
```

Try-catch block

```
try {
    statements . . .
} catch (ExceptionType1 ename1) {
    error handling statements . . .
} catch (ExceptionType2 ename2) {
    error handling statements . . .
} finally {
    … this code always executed …
}
```

Exception Reminder

```
try {
   readFromFile("datafile");
} catch (FileNotFoundException e) {
   System.err.println("Error: File not found");
}
```

Exception Handling: Some Options

- Print something
- Ignore an exception (by having and empty **catch** block)
- Throw a new exception
- Fix the problem
- Exit

Exception Handling: **throw**

- You can `throw` an exception from an exception handler (a `catch` block).
  - Allows you to change exception type and/or error message.

```
catch (ArrayIndexOutOfBoundsException aioobx) {
    throw new ShoppingListTooSmallException(
        "product index is not in the shopping list")
```

Exception Handling: Fix the problem

- You can fix things and then *resume* execution automatically (this is not always possible!)
- You can have a loop the retries the code again.

Exception Handling: exiting

- Sometimes the error is fatal, and you want to stop the program immediately.

```
System.exit();
```

RunTime Exceptions

- There are exceptions that are generated by the system (that are usually caused by programming mistakes):
  - NullPointerException (null references)
  - ArrayIndexOutOfBoundsException
- If you don't catch these, a stack trace will be generated and the program will terminate.
- The compiler does not force you to catch these exceptions.

Create your own Exception Types

```
E.g.,
class myException extends Exception {}

class newException extends Exception {
    newException(){}
    newException(String s) { super(s); }
}

throw new newException("Invalid new");
```