

ICS 111

Introduction to Object Oriented Programming

- Review:
 - variables, values, types, expressions, statements
 - arrays
 - references
 - exceptions
 - conditionals, loops
 - methods
 - I/O
 - evolution of programming languages
- Object-Oriented Programming
- Object-Oriented Design

Variables, Values, Expressions, Types

- Variables in Java are names for memory locations that store (remember) values
- Variables must be declared and initialized
 - variables are initialized to a default value when not initialized explicitly by the programmer
 - constants (introduced with `final`) are variables that we cannot assign to after initialization
 - variables are only valid within their scope, which is from the declaration to the end of the nearest enclosing block
- Expressions compute values
 - we have seen arithmetic and boolean expressions
 - calling a method that returns a value, is an expression
 - anything that computes a value is an expression!
- Variables, values, and expressions are all typed
 - types include the 8 basic types
 - `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`
 - types also include all Object types
- Java is strongly typed: we cannot use an integer or a String where a boolean value is needed
- Java types do offer some flexibility:
 - concatenating anything with a string returns a string, as in `"the value is " + x`
 - in numeric expressions we can combine values with different numeric types
 - each of the 8 basic types is automatically converted to (boxed) and from (unboxed) the corresponding object type
 - object types can also sometimes be used interchangeably (more later)

Statements

- Java execution proceeds one statement at a time, in sequence
 - except when using parallel features such as threads
- there are many kinds of statements:
 - every expression is also a statement
 - just add a semicolon at the end!
 - assignment statements
 - blocks of statements
 - conditionals, switch statements, loops
 - break and return
 - method calls that don't return a value
 - you could say that variable declarations are statements, because they include variable initialization, which is like an assignment
 - you could also say that variable declarations (the ones that don't include initialization) are just declarations, not statements

Arrays and ArrayList

- The programmer has to name and keep track of individual variables
- When we need more data than can be managed in this way, we use arrays or ArrayList
 - these are collections of (potentially large amounts of) data, all of the same type
 - the individual items of data are called elements
- The size of an array is determined by the program when it creates the array
 - the size of an ArrayList varies dynamically
- An index is used to access a single element of an array or ArrayList
 - `al.set(i, a[i]);`
 - in Java, indices always begin with 0
 - the last valid index is `a.length - 1` or `al.size() - 1`
- Loops are essential for accessing and/or modifying the contents of arrays and ArrayList elements

References

- The `new` keyword allocates (reserves) space in memory for a new object, and returns the reference (pointer) to that memory
 - references in Java have the type of the object they refer to
- references can be copied with assignments, and compared with `==`
- copying a reference does not copy the object referred to:
 - to create a new array with contents equal to an old array, you can use `Arrays.copyOf`
 - to create a new string with contents equal to an old string, you can use `new String(oldString)`
- internally to Java, a reference is a special kind of integer, holding the numeric address of the start of the allocated memory
- memory that has been allocated, but that no longer has any references to it, is automatically recycled by the runtime system
 - a process known as garbage collection

Java exceptions

- Java exceptions are two things:
 - a control mechanism, similar to break and return
 - `throwing` an exception transfers control to the nearest enclosing matching `catch` statement
 - enclosing means that this code is executed from within a `try..catch` block
 - for `break`, enclosing refers to the code structure, not the code execution
 - a type of object
 - with constructors that take a string as argument, and also constructors that need no argument
 - exceptions can be printed, assigned to, etc
 - a matching `catch` creates a local variable containing the exception that was `thrown`
- Java exceptions are thrown automatically by many Java built-in operations, including division by zero and using an illegal array index
- If not caught sooner, Java exceptions are caught (and usually printed) by the Java runtime system

Conditionals and Loops

- conditionals provide conditional execution of statements
- loops provide repetition of statements
 - for, enhanced for, while, do...while
- conditionals and loops all evaluate boolean expressions to decide whether to evaluate the next statement or repetition
 - expressions for switch need not be boolean
- writing correct loops include executing:
 - correct code
 - the correct number of times
 - each time with the correct index! (if using an index)

Java methods

- A method (or function) encapsulates some code in a way that is easy to call (invoke)
- methods take parameters:
 - when called, the caller must supply corresponding arguments
- non-void methods return values
 - every branch of the body of the method must return a value of the given type
- methods can call methods, including themselves

Java I/O

- Input and Output includes all communication between the computer and the outside world
- We have seen:
 - getting input from the user
 - printing text or graphics to the display
 - reading or writing files
 - communicating over a network
- I/O also includes:
 - measuring real-world quantities such as temperature
 - turning physical devices on and off
 - regulating the behavior physical devices
 - taking pictures or recording video or audio
- I/O may always fail
 - because the outside world is not as predictable as the CPU
 - I/O failure often leads to throwing an exception

Evolution of Programming Languages: the beginning

- Much of what we've learned so far was available in the earliest programming language, **Fortran**:
 - variables, values, expressions, statements
 - conditionals, loops
 - types: integer and floating point, arrays
 - methods (named functions in Fortran, and many other languages)
- Much was not:
 - characters, strings, ArrayList, Object
 - references, memory management (which in Java we've seen as the `new` keyword)
 - strong typing, meaning a value could be looked at as if it had a different type
 - not having strong typing can be useful if you are thinking in terms of bit representation of values
 - array bounds checks (buffer overflow was ok if done on purpose)
 - exceptions
 - recursive methods
- As soon as Fortran came into being, lots more people and groups created many more languages

Evolution of Programming Languages: Fortran to Java

- As soon as Fortran came into being, lots more people and groups created many more languages
- These languages generally offered features not available in Fortran, including:
 - recursive functions
 - characters and strings, most notably in **COBOL**
 - references, strings, and memory management, notably **C**
 - vector operations (implicit loops to work on an entire array or matrix with a single operation), notably in **APL**
 - functions that could be saved into variables and even customized by pre-specifying some of the parameters, combined with convenient memory management, notably **Lisp, Scheme, ML**
 - more support for structuring code in ways that would be less error-prone
- Support for code structure came in many forms, including modules, which reflect the separation of code into source files
- A popular form of code structuring came in the form of encapsulating data with the methods operating on that data
 - the data is referred to as objects
 - the code is called methods
 - for example, an ArrayList object has an add method. This method only works on ArrayList objects
 - the entire idea was called **Object-Oriented** programming, sometimes abbreviated **OO**
 - **Smalltalk** was an early Object-Oriented programming language
 - Many modern programming languages are Object-Oriented, including notably **Java**

Object-Oriented Programming

- In an Object-Oriented language, every value is an object
 - in Java, all values are objects unless they have one of the 8 basic types
- A **class** defines the type of an object
 - for example, ArrayList is a class, String is a class, Object is a class, ArrayIndexOutOfBoundsException is a class
- Actual object values are **instances** of a class
 - sometimes described as instances of an object
- Each object has some memory allocated for it
 - so that each object can have its own instance variables
 - an instance variable contains the values that are specific to each object
 - for example, two String objects s1 and s2 each have instance variables to store the length of the string
 - and two strings need not have the same length
- Instance variables in Java are usually declared `private`
 - that's why we have to call `s1.length()` instead of using `s1.numChars`
 - `array.length` is the most obvious exception to this rule

Object-Oriented Design

- In an Object-Oriented language, every value is an object
- so most of the effort in designing a program goes to designing the objects
- the type of an object is a **class**
- the variables and methods in a class definition are the variables and methods of objects of that class

Object-Oriented Design Ideas

- The collection of public methods of a class is the **interface** of that class
 - the interface only shows the method headers
 - return type, name, and argument list
 - an interface may include public constants
 - rarely, public variables (such as `array.length`)
- The code of the class, with the private variables, is the implementation of the class
- The caller of a method need only know how to call a method, and what the method does
 - the caller doesn't need to know how a method is implemented
- This means we can change the implementation of a class without impacting other code that calls methods from this class
 - as long as the new implementation does the same as the old!
- For example, internally you could switch from using an array to using an `ArrayList`, or viceversa, and any code that calls the methods of your class would not know the difference

Designing a Class

- Designing a class means designing a collection of methods that are
 - useful, and
 - implementable
- Because objects have private variables that the methods of the class have access to, objects can remember what happened before, and when they are called can modify the private variables
- Example:
 - ArrayList has an internal array of values
 - the add method stores the new value in the array, updating the internal variable that keeps track of the array size
 - the get method returns the corresponding array value
 - but if your code has a variable of type ArrayList, it has access to neither the size variable, nor the array
- The methods together must be sufficient to make objects of that class useful

Designing a Class: Example

- I want to design a method to track my credit card transactions
- useful methods might include:
 - `void recordTransaction(Date date, int amount, String description);`
 - `void paidBill(Date date, int amount);`
 - `int balance();`
 - `void showHistory();`
- private variables might include:
 - an array or `ArrayList` of transactions
 - the current balance
 - a file that tracks the history and balance

Summary

- Review:
 - how to write programs
 - variables, values, expressions, types
 - many kinds of statements
 - methods
- Object-Oriented Programming:
 - some of the effort will always be in coding
 - but coding is easier with useful, well-designed objects