CSC207 Lecture Notes

WEEK 2 (java fundamentals)

1) 8 primitive data types in Java
    a. int (32 bit signed)
    b. byte (8 bit signed)
    c. long (64 bit signed)
    d. float (32 bit single precision)
    e. double (64 bit double precision)
    f. boolean (1 bit)
    g. char (16 bit unsigned)
2) Strings are immutable, i.e. value cannot be changed once created
3) Classes:
    a. Create custom “data types”
    b. Blue print for creating objects
    c. Models the state and behavior of real world object
4) Objects:
    a. Created out of a blue print
    b. Software bundle of related state and behavior
5) Constructors are invoked to create objects from the class blueprint
6) Setters (mutator method): control changes to instance variable
7) Getters (accessor method): return value of instance variable
8) An instance is an object created via the “new” keyword, getters and setters are called instance methods
9) Static fields: call a method when no object of that class has been created yet, i.e. does not depend on instances
10) Local variables:
    a. Declared and visible only in methods or constructors
    b. Created on stack
11) Instance variables:
    a. Declared in class but outside a method or constructor (visible everywhere)
    b. Created on heap
12) Design Pattern 1 (FACTORY METHODS):
    a. Considered as abstraction of a constructor
    b. Factory methods have names and descriptive and static

WEEK 3 (inheritance and composition)

1) Stack:
    a. Stored in computer RAM
    b. Local variables created on stack, automatically deallocate

2) Heap:
    a. Stored in computer RAM
    b. Objects allocated via “new” keyword

    c. Slower allocation compared to stack
    d. Garbage collector deletes variables or objects on heap
    e. No clear “top” item unlike stack
3) Inheritance:
    a. Create a general class that defines traits to a common set of items
    b. “Is-a” relationship
    c. Class that is inherited “SUPER CLASS”
    d. Class that does the inheriting “SUB CLASS”
    e. Variables and methods declared public or protected get inherited
4) Visibility:
    a. Anything that is declared as public is visible in all classes
    b. Anything that is declared as private is only visible in that class
    c. Anything that is declared as protected is visible in that class and all subclasses
    d. Anything that is “no modifier” is visible in that class and package
5) The constructor for superclass constructs the superclass portion of the object
6) The constructor for subclass constructs the subclass portion of the object
7) Parameterized constructor in superclass, use super()
8) Encapsulation separates *implementation (how)* from *behavior (what)*. “hidden”
    a. Hiding the internal details or mechanics of how a class does something
9) Method override: When a method in a subclass has the same return type and signature as a method in the superclass, then the method in the subclass is said to override the method in the superclass
10) Method overload: When a class contains functions with the same name but different input arguments
11) Composition (aggregation): “has-a” relationship
12) Single Responsibility Principle:
    a. Every class should only perform one responsibility or function
    b. And that responsibility should be entirely encapsulated by the class

WEEK 4 (software development process)

1) Always override toString() method for informative representation, pleasant to use
2) Always check parameters for validity:
    a. Method could fail with confusing exception
    b. Method could return normally but compute the wrong result
    c. Method could return normally but leave some objects undetermined
3) WATERFALL MODEL:
    a. Requirements
    b. Design
    c. Implementation
    d. Verification: testing the software // Validation: testing the specifications
    e. Maintenance
    f. Advantages:
        i. Easy to use
        ii. Works well on smaller projects and when requirements are well-understood
        iii. Cost effectiveness

    g. Disadvantages:
        i. Testing
        ii. High risk and uncertainty
        iii. Doesn't work well when requirements change midway
4) ITERATIVE DEVELOPMENT:
    a. In code >> get feedback >> code … cycle
    b. System requirements always evolve in the course of the project
5) INCREMENTAL DEVELOPMENT:
    a. Build as much as you need right now
    b. Each increment delivers part of the required functionality
    c. User requirements are prioritized
6) SPIRAL MODEL:
    a. Determine objectives
    b. Identify and resolve risks: find other approaches to fulfill requirements
    c. Development and Test
    d. Plan the next iteration
7) AGILE METHODS:
    a. Agility is the ability to create and respond to change in order to profit in a turbulent business environment
    b. People oriented
    c. Respond effectively to changes
    d. Creates working systems that meets needs of stakeholders
8) Traditional Timeline: Requirements -> Build -> Test -> Release
9) Agile Timeline: Build & Release -> Build & Release -> Build & Release
    a. Bugs found early
    b. Possible to incorporate feedback
    c. Better visibility of progress
    d. Keep software "releasable"
    e. Lightweight design
    f. Continuous unit testing and refactoring
10) SCRUM AGILE DEVELOPMENT:
    a. Iterative + Incremental
    b. Roles:
        i. Product owner (instructor)
        ii. Team (my group)
        iii. Scrum master: maintains process and enforces rules, responsible for success of project
    c. Product backlog: list of features (user stories)
        i. Users describe situations in which the developed software is to come into play
        ii. User stories should be written by customers
    d. Sprint backlog: subset of features chosen for sprints
        i. Items are drawn from product backlog and broken down into smaller sub-tasks
    e. Code sprint: takes place over weeks

f. Daily scrum: short daily meetings
    i. What did you work on yesterday?
    ii. What will you work on today?
    iii. Is there any issue you are stuck on?
g. Final product: until final product is produced

WEEK 5 (java memory model)

1) Stack region of memory:
    i. Box contains: name of method executing and name of class
    b. Once function returns, local variables are discarded
    c. Only primitives stored on stack
    d. Keeps stack and individual stack frames small
    e. Only references are passed around on the stack
    f. Every new method creates and pushes and new stack frame at the top of the stack and
    g. Removed when the method returns normally
    h. Top most stack frame is being currently executed
    i. LIFO (Last In First Out)
2) Heap region of memory:
    i. Object Heap Space where all the objects/instances live
        1. Box contains: memory address, type of object (class name), instance variables and methods
    ii. Static Heap Space where all the class level (static members) live
        1. Box contains: name of class, name of superclass, static variables and static methods
    b. Objects are created on the heap (“new” keyword)
    c. Removed via the Garbage Collector
    d. Only instance variables and instance methods
    e. Can be shared by multiple threads
3) PRACTICE MEMORY DIAGRAMS

WEEK 6 (polymorphism, interfaces, liskov, singleton)

1) Poly: many + Morph: change
2) Use inheritance for polymorphism and dynamic method binding
3) Enables you to program in general than program in specific
4) Use objects that share the same super class as if they are all objects of the super class
5) Implements systems that are easily extensible
6) We have a base class reference pointing to an object of the subclass
7) Downcasting: cast the superclass reference to a subclass
8) Function binding: mapping from function call to function implementation
    a. Static binding (compile time or early time binding)
        i. Uses the type of the reference
    b. Dynamic binding (runtime time or late time binding)
        i. Uses the class of the object that is referred/pointed to

9) Final methods:
    a. Cannot be overridden in a subclass
    b. Private and static methods implicitly final
10) Final classes:
    a. Cannot be extended by a subclass
    b. All methods in a final class implicitly final
11) Abstract classes:
    a. An abstract class is a placeholder in a class hierarchy that represents a generic concept
    b. Often contains abstract methods, thought it doesn't have to
    c. Abstract methods only contain method declarations and no method body
    d. An abstract class cannot be instantiated
    e. Helps us establish common elements in a class that is too general to instantiate
12) Interface:
    a. Collection of constants and abstract methods
    b. Classifies common traits or behaviors that are exhibited by potentially many non-related classes of objects
    c. A class can implement multiple interfaces but only extend one class (Diamond of Death)
    d. Public visibility
    e. Class that implements interface must define all methods in the interface
13) SINGLETON DESIGN PATTERN:
    a. Only one instance of a class ever be created
    b. Create reference and set to null and all point to one reference
14) LISKOV SUBSTITUTION PRINCIPLE:
    a. Substitutability is the principle of object-oriented programming
        i. If S is a subtype of T then objects of type T may be replaced with objects of type S without altering any of the desirable properties
    b. i.e. we must make sure that new derived classes are extending the base classes without changing their behavior.
    c. In math, square is a rectangle
    d. But, Square Object is not a Rectangle Object because the behavior of Square is not consistent with the behavior of a Rectangle
    e. Change to "has-a" relationship

WEEK 7 (JUnit part 1)

1) Unit testing:
    a. Testing bits of code in isolation with test code
    b. Allows you to make big changes to code quickly
    c. Helps to really understand the design of the code
    d. Helps with code re-use
    e. Gives you instant visual feedback
    f. Help document and define what something is supposed to do

2) Test Suites:
    a. Ad-hoc testing: testing whatever occurs to you at the moment
    b. Test Suite: a thorough set of tests that can be run any time
        i. Disadvantages: extra programming and time consuming
    c. assertEquals(expected value, actual value)
    d. JUnit:
        i. Test fixture: sets up the data needed to run tests
        ii. Unit test: test of a single class
        iii. Test case: tests the response of a single method to a particular set of inputs
        iv. Test suite: collection of test cases
        v. Test runner: software that runs tests and reports results
        vi. Integration test: test of how well classes work together (not well supported by JUnit)
    e. Assert methods:
        i. assertTrue
        ii. assertFalse
        iii. assertEquals
        iv. assertSame
        v. assertNotSame
        vi. assertNull
        vii. assertNotNull
        viii. fail

WEEK 8 (exception and generics)

1) Errors:
    a. Compile time errors: errors caught by the compiler (syntactical error)
    b. Runtime error: errors caught when program is running (insufficient memory)
2) Error handling should be separated in the flow of code from the mainline
3) Have more code to do error detection, reporting and handling
4) Spaghetti code: relation between pieces of code are so tangled that it is nearly impossible modify something without unpredictably breaking something else
5) Exceptions:
    a. Code is easier to read and maintain, more reliable.
    b. Java class that extends Throwable class via “throw” statement
    c. Since it is an alternate return value, it forms part of the signature of the method
    d. Make sure exception handlers are declared in the correct order
6) Set: A collection that contains no duplicate elements (HashSet)
7) Iterator:
    a. Used to traverse through elements of a collection
    b. hasNext(), next(), remove()
8) Generics: Mechanism that could be used to ensure that collections were used safely at compile time. Using “<type>” when declaring collection
9) TreeSet: Industrial-strength version of binary search trees
10) Comparator: pubic interface Comparator<E>, public int compare, obj1.getID

WEEK 9 (iterator and builder)

1) ITERATOR DESIGN PATTERN:
    a. The Iterator pattern allows traversal of the elements of an collection without exposing the underlying implementation
    b. Encapsulation
    c. When different data structures used
    d. Public iterator createIterator();
2) Nested Classes:
        i. Static (declared static): behaviorally a top-level class that has been nested in another top-level class for packaging convenience
            1. OuterClass.staticNestedClass
        ii. Non-static: inner classes
    b. Logical grouping of classes that are only used in one place
    c. Makes package more streamlined
    d. Increases encapsulation. Details of innerclass is not visible to outerclass
    e. Readable and maintainable code
3) BUILDER DESIGN PATTERN:
    a. Use when faced with many constructor parameters
    b. Multiple setter calls puts object in inconsistent state which may cause program failure
    c. Static builder class, if it is non-static then it would require an instance of its owning class.
    d. Always make instances using builder
    e. Advantages:
        i. Calling code is easy to read and write
        ii. Outerclass is immutable

WEEK 10 (publish subscribe)

1) PUBLISH SUBSCRIBE DESIGN PATTERN:
    a. OBSERVER – SUBSCRIBER
    b. OBSERVABLE - PUBLISHER
    c. One or more subscribers are interested in the state of a publisher and register their interest by attaching themselves
    d. A change in our publisher can be notified to all subscribers
    e. When subscriber is not interested in the publisher, they can detach themselves
    f. Advantages:
        i. Break down applications into smaller, more loosely coupled modules, which can improve general manageability
        ii. Think hard about different parts of the application, i.e. we can identify which parts act as publishers and which as subscribers
        iii. Best tool for designing decoupled systems

WEEK 11 (regular expressions)

1) Regular expression:
    a. Specific kind of text pattern
    b. Use with many modern programming languages
    c. Create matchers and patterns to use regular expressions
    d. Very input string fits into text pattern
    e. Find text that matches the regex pattern
    f. Extract or Substitute text; useful for manipulating text
    g. Used in automatic generation of webpages
    h. Greedy quantifier: match as much as it can
    i. Reluctant quantifier: match as little as possible, add "?" to make reluctant
    j. Possessive quantifier: match as much as it can and never let go, add "+" to make possessive
2) Matcher methods:
    a. m.matches(): returns true if pattern matches entire text
    b. m.lookingAt(): returns true if pattern matches beginning of text
    c. m.find(): returns true if pattern matches any part of text
    d. m.start(): returns index of first character matched
    e. m.end(): returns index of last character matched plus one
    f. m.replaceFirst(replacement): returns new string where first substring matched is replaced by replacement
    g. m.replaceAll(replacement): returns new string where every substring matched is replaced by replacement
    h. m.find(startIndex): looks for next match starting at start index
    i. m.reset(): resets the matcher
    j. m.group(n): returns the string matched by the capturing group n
    k. m.group(): returns the string matched by the entire pattern (m.group(0))
3) Capture groups:
    a. \0: entire matched string, \1 first matched group, \2 second matched group
    b. capturing groups are numbered by counting their opening parentheses from left to right.
4) Types of Regex:
    a. [a-z]+: match a sequence of one or more lowercase letters
    b. immediately after open bracket, "^" means "not"
    c. [a-zA-Z0-9]: any one letter or digit
    d. "|": or
    e. ".": any one character
    f. \d: a digit [0-9]
    g. \D: a non-digit [^0-9]
    h. \s: a whitespace [ ]
    i. \S: a non-whitespace [^\s]
    j. \w: a word character [a-zA-Z_0-9]
    k. \W: a non-word character [^\w]
    l. "^" matches beginning of line
    m. "$" matches end of line

n. \b: a word boundary
o. \B: not a word boundary
p. "?": optional
q. "*": 0 or more times
r. "+": 1 or more times
s. "{n}": occurs n times
t. "{n,}": occurs n or more times
u. "{n, m}": occurs at least n but not more than m times
v. (.*): all the rest of the characters
w. match meta-characters using "\"

WEEK 12 (JUnit2 and refactoring)

1) Test Driven Development (TDD):
    a. Traditional Design: Design -> Code -> Test
    b. TDD: Design -> Test -> Code
    c. Result changes the quality of work
    d. Forces us to think about module thinking from outside looking in rather than inside looking out
    e. Break requirements down into very small units of testable functionality
    f. How to test resulting code through its public interface
    g. Design for implementation becomes a secondary concern
    h. Relies on the repetition of a very short development cycle:
        i. Developer writes an (initially failing) automated test that defines a new function
        ii. Then produces the minimum amount of code to pass that test -> refactors code to acceptable standards
        iii. Test the tests. Run the new tests to verify they fail
        iv. Write code -> Rerun tests to verify that they now succeed -> Refactor -> Repeat
2) Stubs:
    a. Minimal methods that always return the same values
    b. When we run tests with stubs, we want the test methods to fail
    c. This helps "test the tests" so that incorrect methods don't pass the test
3) Mock Objects:
    a. Create and instantiate fake objects for other classes so that you fulfill the obligations of your class that is currently under test
    b. Use live googleScholarPages as mock objects for Assignment 3

WEEK 12 (introduction to refactoring)

1) Refactoring:
    a. Restructuring (rearranging) code in a series of small, working code, in order to make the code easier to maintain and modify
    b. Unit test to prove that code still works
    c. Code is more loosely coupled, more cohesive modules, more comprehensible

    d. Refactoring “catalog”
2) When to refactor:
    a. Any time you see a better way of doing things
    b. Without breaking the code
    c. Should not refactor stable code that won’t change, someone else’s code
3) “bad smell”: an indication that something is wrong
4) When can you refactor:
    a. You should be in a supportive environment
    b. You are familiar with common refactorings
    c. Adequate set of unit tests
5) Refactoring process:
    a. Make a single refactoring
    b. Run all tests to ensure everything works
    c. Then move to next refactoring
    d. If not, fix the problem or undo the change
6) Code smells:
    a. Duplicate code
    b. Long methods
    c. Big classes
    d. Big switch statements
    e. Long navigations
    f. Lots of checking for null objects
    g. Data clumps
    h. Data classes
    i. Un-encapsulated fields
7) Techniques:
    a. Switch statements
    b. Encapsulate fields
    c. Extract class: break one class into two, each with appropriate responsibility
    d. Extract interface: extract an interface from a class; more specialized interfaces than one multi-purpose interface
    e. Extract method: most useful tool for reducing amount of duplication in code
    f. Extract subclass: create a specialization of that class and give it features that would only be useful in specialized instances. Good design is binding to abstractions whenever possible
    g. Extract superclass: if 2 or more classes have shared features, abstract those shared features into a superclass
    h. Move method – before: if a method on one class uses another class more than the class on which it is defined, move it to the other class.
    i. Replace error code with exception: a method that returns special code to indicate an error is better accomplished with an exception
8) Code reviews:
    a. A constructive review of a fellow developer’s code. A required sign-off from another team member before a developer is permitted to check in changes or new code
    b. Maintainable, dry, readable and bug-free code

9) Mechanics of code review:
    a. Who: original developer and reviewer
    b. What: reviewer gives suggestions for improvement on a logical and/or structural level. Feedback leads to refactoring
    c. When: when code author has finished a coherent system change that is otherwise ready for check-in
    d. Advantages:
        i. Increases quality threshold
        ii. Forces code authors to articulate their decisions
        iii. Hands-on learning experience for rookies without hurting code quality
        iv. Team members involved in different parts of the system
        v. Author and reviewer both accountable for committing code

==========