SOFTWARE CODING

Software Engineering

Dr. Raj Singh

CODING CONCEPTS

BEFORE YOU CODE

Confirm the detailed designs you must implement	code only from design
Prepare to measure time spent, classified by	detailed design; design review; coding; coding review; compiling & repairing syntax defects; unit testing & repairing defects found in testing
Prepare to record defects using a form	default: major (requirements unsatisfied), trivial, or neither default: error, naming, environment, system, data, other
Understand required standards	for coding
Estimate size and time based on your past data	
Plan the work in segments	

WRITING CODE

- Plan the structure and residual design for your code
- Self-inspect your design
- Write your code
- Self-inspect your code
- Test your code
- Compile your code
- Fix the issues don’t ignore it

GENERAL PRINCIPLES IN PROGRAMMING PRACTICE

Try to reuse first

Enforce intentions

If code is intended to be used in particular ways only, write it so that the code cannot be used in any other way

APPLICATIONS OF “ENFORCE INTENTIONS”

If a member is not intended to be used by other functions, enforce this by making it private or protected.

Use qualifiers such as final and abstract etc. to enforce intentions

THINK GLOBALLY, PROGRAM LOCALLY

Make all members as local as possible

And as invisible as possible

access private members through public accessor functions if required.

IMPLEMENT EXCEPTION & ERROR HANDLING

- Follow agreed-upon development process
- Catch many errors at compile-time
- Where error handling is specified by requirements, implement as required
- Anticipate all possible implementation defects
- Follow a consistent policy for checking parameters
- Rely mostly on good design and development process
- Treat warnings as errors and don’t ignore it

PROGRAMMING STANDARDS

NAMING STANDARDS

- Use concatenated words
  - e.g. cylinderLength
- Begin class names with capitals
  - e.g. Shapes
- Variable names begin lower case
  - e.g. cylinderlength
- Constants with capitals
  - e.g. MAX_NAME_LENGTH
- Data members of classes with an underscore
  - e.g. _timeOfDay
- Use get, set, and is for accessor methods
  - e.g. getName(), setName(), isBox()

DOCUMENTING METHODS

- What the method does?
- What parameters it must be passed (use @param tag)
- What does it return?
- Exceptions it throws (use @exception tag)

ATTRIBUTES

- Description -- what it's used for
- All applicable invariants
  - quantitative facts about the attribute,
  - such as "1 < _age < 130"
  - or " 36 < _length * _width < 193".

CONSTANTS

- If certain attribute values will never change
  - Define as constants
  - Use static final

OBJECT ORIENTED PROGRAMMING

OBJECT-ORIENTED PROGRAMMING/DESIGN (OOP/OOD)

A paradigm that represents concepts as "objects" that have attributes that describe the object and associated procedures known as methods

Objects, which are usually instances of classes, are used to interact with one another to design applications

Objective-C, Smalltalk, Java and C# are few examples of object-oriented programming languages

OBJECT AND CLASS

An object can be considered as a "thing" that can perform a set of related activities.

The set of activities that the object performs defines the object's behavior.

In pure OOP terms an object is an instance of a class.

OBJECT AND CLASS

A class is a representation of a type of object.

It describe the details of an object.

Class is composed of three things: name, attributes, and operations.

```
public class Student {
    private String name; // attributes
    …
    private void method1(){ // operations
      …
    }
  …
}
Student objectStudent = new Student();
```

student object, named objectStudent, is created out of the Student class.

CLASS DESIGN PRINCIPLE

SRP - The Single Responsibility Principle	A class should have one, and only one, reason to change.
OCP - The Open Closed Principle	You should be able to extend a classes behavior, without modifying it.
LSP - The Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
DIP - The Dependency Inversion Principle	Depend on abstractions, not on concretions.
ISP - The Interface Segregation Principle	Make fine grained interfaces that are client specific

MAIN OOP/OOD CONCEPTS

Encapsulation	information hiding
Abstraction	define, don't implement
Inheritance	extensibility
Polymorphism	one object many shapes

ABSTRACTION AND GENERALIZATION

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars.

"What" rather than "How"

Generalization is the broadening of application to encompass a larger domain of objects of the same or different type.

Abstraction and generalization are often used together.

INHERITANCE

Software reusability

Reuse an existing class and it's behavior

Create new class from an existing class

Absorb existing class's data and behaviors

Enhance with new capabilities

Subclass extends superclass

More specialized group of objects

Behaviors inherited from superclass

ABSTRACT CLASSES AND METHODS

- Classes that are too general to create real objects
- Used only as abstract superclasses for concrete subclasses and to declare reference variables
- Many inheritance hierarchies have abstract superclasses occupying the top few levels
- Keyword abstract
  - Use to declare a class abstract
  - Also use to declare a method abstract
- Abstract classes normally contain one or more abstract methods
- All concrete subclasses must override all inherited abstract methods

INTERFACES

- Interfaces are used to separate design from coding as class method headers are specified but not their bodies.
- Interfaces are similar to abstract classes but all methods are abstract and all properties are static final.
- Interfaces can be inherited (i.e.. you can have a sub-interface).
- An interface is used to tie elements of several classes together.
- This allows compilation and parameter consistency testing prior to the coding phase.
- Interfaces are also used to set up unit testing frameworks.

POLYMORPHISM

Facilitates adding new classes to a system with minimal modifications

When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable

The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked

TYPES OF POLYMORPHISM

Overloading

More than one method in a class with same name different signature.

Does not depend on return type.

Overriding

Method in a subclass with same name and return type.

Dynamic Binding

Also known as late binding

Calls to overridden methods are resolved at execution time, based on the type of object referenced

VOCABULARY

VOCABULARY I

- class – a description of a set of objects
- object – a member of a class
- instance – same as “object”
- field – data belong to an object or a class
- variable – a name used to refer to a data object
  - instance variable – a variable belonging to an object
  - class variable, static variable – a variable belonging to the class as a whole
  - method variable – a temporary variable used in a method

VOCABULARY II

- method – a block of code that can be used by other parts of the program
  - instance method – a method belonging to an object
  - class method, static method – a method belonging to the class as a whole
- constructor – a block of code used to create an object
- parameter – a piece of information given to a method or to a constructor
  - actual parameter – the value that is passed to the method or constructor
  - formal parameter – the name used by the method or constructor to refer to that value
- return value – the value (if any) returned by a method

VOCABULARY III

- hierarchy – a treelike arrangement of classes
- root – the topmost thing in a tree
- Object – the root of the class hierarchy
- subclass – a class that is beneath another in the class hierarchy
- superclass – a class that is above another in the class hierarchy
- inherit – to have the same data and methods as a superclass