Programming Concepts Using Java

Revision Slides

W01:L01: Introduction

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, ...
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there
- ... and why new ones are still being created

W01:L02: Types

- Types have many uses
  - Making sense of arbitrary bit sequences in memory
  - Organizing concepts in our code in a meaningful way
  - Helping compilers catch bugs early, optimize compiled code
- Some languages also support automatic type inference
  - Deduce the types of a variable statically, based on the context in which they are used
  - `x = 7` followed by `y = x + 15` implies `y` must be `int`
  - If the inferred type is consistent across the program, all is well

W01:L03: Memory Management

- Variables have **scope** and **lifetime**
  - Scope — whether the variable is available in the program
  - Lifetime — whether the storage is still allocated
- Activation records for functions are maintained as a stack
  - Control link points to previous activation record
  - Return value link tells where to store result
- Two ways to initialize parameters
  - Call by value
  - Call by reference
- Heap is used to store dynamically allocated data
  - Outlives activation record of function that created the storage
  - Need to be careful about deallocating heap storage
  - Explicit deallocation vs automatic garbage collection

W01:L04: Abstraction and Modularity

- Solving a complex task requires breaking it down into manageable components
  - Top down: refine the task into subtasks
  - Bottom up: combine simple building blocks
- Modular description of components
  - Interface and specification
  - Build prototype implementation to validate design
  - Reimplement the components independently, preserving interface and specification
- PL support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming

W01:L05: OOPS

- Objects are like abstract datatypes
- Uniform way of encapsulating different combinations of data and functionality
- Distinguishing features of object-oriented programming
  - Abstraction
    - Public interface, private implementation, like ADTs
  - Subtyping
    - Hierarchy of types, compatibility of interfaces
  - Dynamic lookup
    - Choice of method implementation is determined at run-time
  - Inheritance
    - Reuse of implementations

W01:L06: Classes

- A class is a template describing the instance variables and methods for an abstract datatype
- An object is a concrete instance of a class
- We should separate the public interface from the private implementation
- Hierarchy of classes to implement subtyping and inheritance
- A language like Python has no mechanism to enforce privacy etc
  - Can illegally manipulate private instance variables
  - Can introduce inconsistencies between subtype and parent type
- Use strong declarations to enforce privacy, types
  - Do not rely on programmer discipline
  - Catch bugs early through type checking

Getting started with Java

- Java program to print hello, world

```
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println("hello, world);
    }
}
```

- A Java program is a collection of classes
- All code in Java lives within a class
- Modifier public specifies visibility
- The signature of main( )
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is void
- Write once, run anywhere

Scalar types

- Java has eight primitive scalar types
  - int, long, short, byte
  - float, double
  - char
  - boolean
- We declare variables before we use them

```
int x, y;
x = 5;
y = 10;
```

- Characters are written with single-quotes (only)

```
char c = 'x';
```

- Boolean constants are true, false

```
boolean b1, b2;
b1 = false;
b2 = true;
```

Scalar types

- Initialize at time of declaration

```
flat pi =  3.1415927f;
```

- Modifier final indicates a constant

```
final float pi = 3.1415927f;
```

Operators

- Arithmetic operators are the usual ones

  `+, -, *, /, %`

- No separate integer division operator //
- When both arguments are integer, / is integer division
- No exponentiation operater, use Math.pow()
- Math.pow(a,n) returns a^n
- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++; // Same as a = a+1
b--; // Same as b = b-1
```

- Shortcut for updating a variable

```
int a = 0, b = 10;
a += 7; // Same as a = a+7
```

Strings

- String is a built-in class
- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- + is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
// "Hello world"
```

- Strings are not arrays of characters
- Instead use s.charAt(0), s.substring(0,3)

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

- Or int a[] instead of int[] a
- a.length gives size of a
- Array indices run from 0 to a.length-1

Control flow

- Conditional execution

```
if (condition) { ... } else { ... }
```

- Conditional loops

```
while (condition) { ... }
do { ... } while (condition)
```

- Iteration - Two kinds of for
- Multiway branching – switch

Classes and objects

- A class is a template for an encapsulated type
- An object is an instance of a class

```
public class Date {
    private int day, month, year;
    public Date(int d, int m, int y){
        day = d;
        month = m;
        year = y;
    }
    public int getDay(){
        return(day);
    }
}
```

- Instance variables - Each concrete object of type Date will have local copies of date, month, year

Creating and initializing objects

- new creates a new object
- How do we set the instance variables?
- Constructors — special functions called when an object is created
  - Function with the same name as the class
  - d = new Date(13,8,2015);
- Constructor overloading - same name, different signatures
- A constructor can call another one using this
- If no constructor is defined, Java provides a default constructor with empty arguments
  - new Date() would implicitly invoke this
  - Sets instance variables to sensible defaults
  - For instance, int variables set to 0
  - Only valid if no constructor is defined
  - Otherwise need an explicit constructor without arguments

Copy constructors

- Create a new object from an existing one

```
public class Date {
    private int day, month, year;
    public Date(int d, int m, int y){
        day = d; month = m; year = y;
    }
    public Date(Date d){
        this.day = d.day; this.month = d.month; this.year = d.year;
    }
}
public class UseDate() {
    public static void main(String[] args){
        Date d1,d2;
        d1 = new Date(12,4,1954); d2 = new.Date(d1);
    }
}
```

Basic input and output in java

- Reading input
  - Use Console class
  - Use Scanner class

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
int age = in.nextInt();
```

W03:L01: The philosophy of OO programming

- Structured programming
  - The algorithms come first
    - Design a set of procedures for specific tasks
    - Combine them to build complex systems
  - Data representation comes later
    - Design data structures to suit procedural manipulations
- Object Oriented design
  - First identify the data we want to maintain and manipulate
  - Then identify algorithms to operate on the data
- **Designing objects**
  - **Behaviour** – what methods do we need to operate on objects?
  - **State** – how does the object react when methods are invoked?
    - State is the information in the instance variables
    - **Encapsulation** – should not change unless a method operates on it

W03:L01: The philosophy of OO programming (Cont.)

- Relationship between classes
  - Dependence
    - Order needs Account to check credit status
    - Item does not depend on Account
    - Robust design minimizes dependencies, or coupling between classes
  - Aggregation
    - Order contains Item objects
  - Inheritance
    - One object is a specialized versions of another
    - ExpressOrder inherits from Order
    - Extra methods to compute shipping charges, priority handling

W03:L02: Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclass can add more instance variables and methods
  - Can also override methods
- Subclasses cannot see private components of parent class
- Use super to access constructor of parent class
- Manager objects inherit other fields and methods from Employee
- Every Manager has a name, salary and methods to access and manipulate these.

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism

- Manager can redefine bonus()

```
double bonus(float percent){
  return 1.5*super.bonus(percent);
}
```

  - Uses parent class bonus() via super
  - Overrides definition in parent class
- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke e.setSecretary()?
  - e is declared to be an Employee
  - Static typechecking – e can only refer to methods in Employee

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

- What about e.bonus(p)? Which bonus() do we use?
  - Static: Use Employee.bonus()
  - Dynamic: Use Manager.bonus()
- **Dynamic dispatch** (dynamic binding, late method binding, . . .) turns out to be more useful
- **Polymorphism**
  - Every Employee in emparray "knows" how to calculate its bonus correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);
emparray[0] = e;
emparray[1] = m;
for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

- Signature of a function is its name and the list of argument types
- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - Employee.bonus()
  - Manager.bonus()
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
    // sorts contents of darr
Arrays.sort(iarr);
    // sorts contents of iarr
class Arrays{
    ...
    public static void sort(double[] a){..}
        // sorts arrays of double[]
    public static void sort(int[] a){..}
        // sorts arrays of int[]
    ...
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`
- e.setSecretary() does not work
  - Static type-checking disallows this
- Type casting — convert e to Manager

  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if e is not a Manager
- Can test if e is a Manager

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L04: The Java class hierarchy

- Java does not allow multiple inheritance
  - A subclass can extend only one parent class
- The Java class hierarchy forms a tree
- The root of the hierarchy is a built-in class called Object
  - Object defines default functions like equals() and toString()
  - These are implicitly inherited by any class that we write
- When we override functions, we should be careful to check the signature
- Useful methods defined in Object

```
public boolean equals(Object o) // defaults to pointer equality
public String toString()        // converts the values of the
                                // instance variables to String
```

- For Java objects x and y, x == y invokes x.equals(y)
- To print o, use System.out.println(o+"");
  - Implicitly invokes o.toString()

W03:L05: Subtyping vs inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If B is a subtype of A, wherever we require an object of type A, we can use an object of type B
  - Employee e = new Manager(...); is legal
  - Compatibility of interfaces
- Inheritance
  - Subtype can reuse code of the main type
  - B inherits from A if some functions for B are written in terms of functions of A
  - Manager.bonus() uses Employee.bonus()
  - Reuse of implementations
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two

W03:L06: Java modifiers

- private and public are natural artefacts of encapsulation
  - Usually, instance variables are private and methods are public
  - However, private methods also make sense
- Modifiers static and final are orthogonal to public/private
- Use private static instance variables to maintain bookkeeping information across objects in a class
  - Global serial number, count number of objects created, profile method invocations, . . .
- Usually final is used with instance variables to denote constants
- A final method cannot be overridden by a subclass
- A final class cannot be inherited
- Can also have private classes

Abstract classes

- Sometimes we collect together classes under a common heading
- Classes Swiggy, Zomato and UberEat are all food order
- Create a class FoodOrder so that Swiggy, Zomato and UberEat extend FoodOrder
- We want to force every FoodOrder class to define a function
  `public void order() {}`
- Now we should force every class to define the public void order();
- Provide an abstract definition in FoodOrder
- public abstract void order();

Interfaces

- An interface is a purely abstract class
- All methods are abstract by default
- All data members are final by default
- If any class implement an interface, it should provide concrete code for each abstract method
- Classes can implement multiple interfaces
- Java interfaces extended to allow static and default methods from JDK 1.8 onwards
- If two interfaces has same default/static methods then its implemented class must provide a fresh implementation
- If any class wants to extend another class and an interface then it should inherit the class and implements interface

private classes

- An instance variable can be a user defined type

```
public class BookMyshow{
      String user;
      int tickets;
      Payment payement;
}
public class Payment{
     int cardno;
     int cvv;
}
```

- Payment is a public class, also available to other classes
- Payment class has sensitive information, so there is a security concern.

private classes

- We cannot declare Payment class as private outside the BookMyshow class
- You can declare Payment class as private inside the BookMyshow class

```
public class BookMyshow{
      String user;
      int tickets;
      Payment payement;
      private class Payment{
         int cardno;
         int cvv;
      }
}
```

- Now Payment class is a private member of the BookMyshow class
- Now Payment class only available to the BookMyshow class

Interaction with State(Manipulating objects)

- Consider the class student below.
- Student class is encapsulated by private variables.

```
public class Student{
      private String rollno;
      private String name;
      private int age;
      //3 mutator methods
      //3 Accessor methods
}
```

- Consider Student class has student1,student2.....student60 objects
- Update date as a whole, rather than individual components

Interaction with State(Manipulating objects)


```
public class Student{
      private String rollno;
      private String name;
      private int age;
      public void setStudent(String rollno,String name,int age){
      }
}
```

- Now public void setStudent(String rollno, String name, int age) update the Student object as a whole.

Java Call back methods.

- what is call back method?

```
interface Notification{
void notification();//should be overridden in WorkingDay and Weekend
}
class WorkingDay implements Notification{
}
class Weekend implements Notification{
}
class Timer{//Timer will decide which call back function should be call
}
public class User {
    public static void main(String[] args) {
        Timer timer=new Timer();
        timer.start(new Date());
    }
}
```

Iterators

- what is Iterator?
- You can loop through any data structure using an Iterator.

```
public interface Iterator{
public abstract boolean has_next();
public abstract Object get_next();
}
```

W5:L1: Polymorphism Revisited

- In object-oriented programming, polymorphism usually refers to the effect of **dynamic dispatch**
- Depending upon the object type stored in a reference variable appropriate version of overridden and non-overridden methods are invoked automatically.
- **Structural Polymorphism**
  - ```
    public int find(Superclass[] arr, subclass o){
        int i;
        for(i =0 ;i < arr.length; i++){
            if(arr[i].equals(o)) return i;
        }
        return -1;
    }
    ```
  - *Can also be obtained by using interfaces, this is what we have been doing with* **Comparable**<**T**> *interface*
- We are actually **grouping types with one common behaviour** under a parent type (class/interface) which can then polymorphically refer to appropriate subtypes depending upon actual instance type.
    ```

W5:L1: Polymorphism Revisited (Cont.)

- **Type Consistency**:
  Source type can be either same as target type or a subtype of target type. In other words a super type reference variable/array can store subtype objects but not vice versa.
- **Inference**:
  Inheritance/polymorphism cannot guarantee a complete **type generalization** of our program.
- **Using *Object* to generalize types in a program**
  Problems:
  - Type information is lost needs explicit casting on every use.
  - Homogeneity cannot be guaranteed.
- **Solution: Generics**
  - Classes and functions can have type parameters
    1. `class MyDataStructure<T> holds values of type T`
    2. `public T getMatch(T obj) accepts and returns values of same type T as enclosing enclosing class`
  - Can also use constraints by mixing inheritance rules.

```
public static <S extends T,T> void getMatch(S[] sarr, T obj){...}
```

W5:L2: Generics

- Example of a polymorphic List

```
public class LinkedList<T>{
    private int size;
    private Node first;
    public T head(){
        T returnval;
        ...
        return(returnval);
    }
    public void insert(T newdata){...}
    private class Node {
        private T data;
        private Node next;
        ...
    }
}
```

W5:L2: Generics (Cont.)

- Be careful not to accidentally hide a type variable

```
class Myclass<S>{
    public <S,T> void myMethod(S obj){
        T obj2;
        ...
    }
}
```

Quantifier <S,T> of myMethod masks the type parameter S of class MyClass.

W5:L3: Generic Subtyping

- **Covariance of types:**
  If S is a subtype of T and a reference of T can store an object of S, then T[] can also refer S[].
  Arrays are covariant:
  ```
  Integer[] arr1 = {10,20,30};
  Number[] arr2 = arr1;
  System.out.println(arr2[2]); // 30
  ```
- Now, try running this:
  ```
  arr2[1] = 9.8; // It's not allowed, generates exception.
  ```
  The detailed type checking is only done **only at runtime**, *compiler will check for supertype-subtype relation and if that conforms, it will allow the code.*
  Why the statement is executing at runtime?
- **Issue with generics:**
  JVM erases the type information related to a generic type after the compilation is done, i.e. at runtime unlike non-generic type variables/references, generic variables will not have any type characteristics. This process is called **type erasure.**
  Which means all type checking must be done by compiler, but as compiler cannot check object's type during compile time, so JAVA prohibits the covariance property for generic types.
- `List<Subtype>` is not compatible with `List<Supertype>`
  ```
  List<String> s = {"A","B"};
  List<Object> o = s;  Illegal use
  ```

W5:L3: Wildcard

- We can solve this problem using wildcards <?>.
- Avoid unnecessary type quantification when type variable is not needed elsewhere.
- Beneficial while comparing two different subtypes of a common supertype. [1]
- **Bounded Wildcards**

```
LinkedList<?  extends T>
LinkedList<?  super T>
```

[1] will discuss the program.

W05:L04:Reflection

Reflective programming or *reflection* is the ability of a process to examine, introspect, and modify its own structure and behaviour. (Source: Wikipedia)

- Introspect: A program can observe, and therefore reason about its own state.
- Intercede: A program can modify its execution state or alter its own interpretation or meaning.

Reflection in Java


```
Employee e = new Manager(...);
...
if (e instanceof Manager){
   ...
}
```

- What if we don't know the type that we want to check in advance?
- Suppose we want to write a function to check if two different objects are both instances of the same class?

Reflection in Java . . .


```
public static boolean classequal(Object o1, Object o2){
  ...
  // return true iff o1 and o2 point to objects of same type
  ...
}
```

- We cannot use `instanceof` because we will have to check across all defined classes, which is not a fixed set.
- We cannot use generic type variables becaue `if (o1 instance of T)` is not permitted.

Introspection in Java

- Can extract the class of an object using `getClass()`
- `getClass()` returns an object of type `Class` that encodes class information

```
import java.lang.reflect.*;
class MyReflectionClass{
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

Using the `Class` object

- Can create new instances of a class at runtime

```
Class c = obj.getClass();
Object o = c.newInstance();
// Create a new object of same type as obj
```

- Can also get hold of the class object using the name of the class

```
String s = "Manager".
Class c = Class.forName(s);
Object o = c.newInstance();
```

- ..., or, more compactly

```
Object o = Class.forName("Manager").newInstance();
```

The class `Class` ...

- From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- Constructors, methods and fields themselves have structure
  - Constructors: arguments
  - Methods : arguments and return type
  - All three: modifiers `static`, `private` etc
- Additional classes `Constructor`, `Method`, `Field`
- Use `getConstructors()`, `getMethods()` and `getFields()` to obtain constructors, methods and fields of `C` in an array.

The class `Class` ...

- Extracting information about constructors, methods and fields

```
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Method[] methods = c.getMethods();
Field[] fields = c.getFields();
```

- `Constructor`, `Method`, `Field` in turn have functions to get further details
- Example: Get the list of parameters for each constructor

```
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
  Class params[] = constructors[i].getParameterTypes();

}
```

Reflection and security

- Can we extract information about private methods, fields,. . . ?
- For private methods, fields:

```
Constructor[] constructors = getDeclaredConstructors();
Method[] methods = getDeclaredMethods();
Field[] fields = getDeclaredFields();
```

- Security issue : Access to private components may be restricted through external security policies
- To be used sparingly

W5:L5: Type Erasure

- Java does not keep type information of generics at runtime, all type compatibilities are checked during compile time.
  `if(s instanceof ArrayList<String>)` // Compilation error
- At run time, all type variables are promoted to `Object`
  `ArrayList<T> becomes ArrayList<Object>`
- Or the upper bound,if available
  `ArrayList<T extends Mammals> becomes ArrayList<Mammals>`
- Type erasure leads to illegal overloading which were legal on non generics.
  ```
  public void myMethod(ArrayList<Integer> i)...
  public void myMethod(ArrayList<Mammal> m)...
  ```
- To avoid runtime errors generic type arrays can be declared but can't be instantiated.
  ```
  T[] arr;
  arr = new T[20];  // Compiler error
  ```

W06:L01: Indirection

- Suppose two separate implementation

```
public class CircularArrayQueue<E> {
public void add (E element){...};
public E remove(){...};
public int size(){...};
...
}

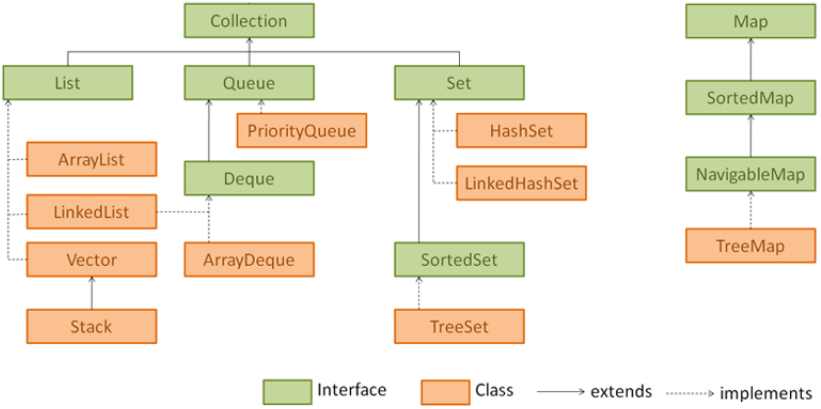
public class LinkedListQueue<E> {
public void add (E element){...};
public E remove(){...};
public int size(){...};
...
}
```

W06:L02: Java-Collections

- Collections Framework?
  - collection of interfaces and classes.
  - organizing a group of heterogeneous objects efficiently.
  - Framework has several useful classes which have tons of useful methods which makes a programmer task super easy.
  - Some collections allow duplicate elements, while others do not.
  - Some collections are ordered and others are not.
  - Reduced development effort by using core collection classes rather than implementing our own collection classes.

W06:L02: Java-Collections

https://www.sneppets.com/java/collections-framework/collection-and-collections-framework-in-java/



W06:L03: Java-Concrete-Collections

- The List interface
  - An ordered collection can be accessed in two ways
  - Through an iterator
  - By position
- ```
  public interface List<E>
  extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);
  }
  ```
- List interface implemented classes.
  - ArrayList
  - LinkedList
  - Vector
  - Stack
  ```

W06:L03: Java-Concrete-Collections

- The Set interface
  - A set is a collection without duplicates.
  - We cannot predict the insertion order.
- Set interface implemented classes.
  - HashSet
  - TreeSet
- The Queue interface
  - Ordered, remove front, insert rear.
- Queue interface implemented classes.
  - ArrayDeque
  - PriorityQueue

W06:L04: Maps

- The Map interface
  - Key-value structures come under the Map interface.
  - Two type parameters
  - K is the type for keys
  - V is the type for values
- Map interface implemented classes.
  - HashMap
  - TreeMap
  - LinkeHashMap

W07:L01: Dealing with errors

- Our code could encounter many types of errors
  - *User input* — enter invalid filenames or URLs
  - *Resource limitations* — disk full
  - *Code errors* — invalid array index, key not present in hash map, refer to a variable that is null, divide by zero, . .
- When we could anticipate what is going to happen we would rather signal the error than program crash
- Exception handling - gracefully recover from errors that occur when running code

W07:L01: Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - RunTimeException, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
  - Array index out of bounds, invalid hash key, . . .
- `Checked exceptions`
  - Typically user-defined, code assumptions violated
  - In a list of orders, quantities should be positive integers

W07:L02: Catching and handling exceptions

- try-catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Use try-catch to safely call functions that may generate errors
- If try encounters an exception, rest of the code in the block is skipped
- If exception matches the type in catch, handler code executes
- Otherwise, uncaught exception is passed back to the code that called this code
- Can catch more than one type of exception
  - Multiple catch blocks
- Catch (ExceptionType e) matches any subtype of ExceptionType
- Catch blocks are tried in sequence
  - Match exception type against each one in turn
- Order catch blocks by argument type, more specific to less specific

W07:L02: Throwing exceptions

- Declare exceptions thrown in header
- Can throw multiple types of exceptions
- Can throw any subtype of declared exception type
  - Can throw FileNotFoundException, EOFException, both subclasses of IOException
- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception
- Customized exceptions - Define a new class extending Exception
- Cleaning up resources
  - When exception occurs, rest of the try block is skipped
  - May need to do some clean up (close files, deallocate resources, . . .)
  - Add a block labelled finally

W07:L03: Packages

- Java has an organizational unit called package
- Can use import to use packages directly
- If we omit modifiers, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility with respect to inheritance hierarchy
  - protected means visible within subtree, so all subclasses
  - Normally, a subclass cannot expand visibility of a function
  - However, protected can be made public

W07:L04: Assertions

- Assertion checks are supposed to flag fatal, unrecoverable errors
- This should not be caught – Abort and print diagnostic information (stack trace)
- If assertion fails, code throws `AssertionError`

```
public static double myfn(double x){
    assert x >= 0;
}
```

- Can provide additional information to be printed with diagnostic message

```
public static double myfn(double x){
    assert x >= 0 : x;
}
```

- If you need to flag the error and take corrective action, use exceptions instead
- Turned on only during development and testing
  - Not checked at run time after deployment

W07:L04: Assertions (Cont.)

- Assertions are enabled or disabled at runtime – does not require recompilation
- Use the following flag to run with assertions enabled

  ```
  java -enableassertions MyCode
  ```
- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class

  ```
  java -ea:Myclasdes MyCo
  ```
- ... or a package

  ```
  java -ea:in.ac.iitm.onlinedegree MyCode
  ```
- Similarly, disable assertions globally or selectively

  ```
  java -disableassertions MyCode
  java -da:MyClass MyCode
  ```
- Can combine the two

  ```
  java -ea in.ac.iitm.onlinedegree -da:MyClass MyCode
  ```
- Separate switch to enable assertions for system classes

  ```
  java -enablesystemassertions MyCode
  java -esa MyCode
  ```

W07:L05: Logging

- Logging gives us more flexibility and control over tracking diagnostic messages than simple print statements
- Example: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- Can define a hierarchy of loggers
- Seven levels of messages — `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`
  - By default, first three levels are logged
- Can set a different level

```
logger.setLevel(Level.FINE);
```

- Turn on all levels, or turn off all logging

```
logger.setLevel(Level.ALL);
logger.setLevel(Level.OFF);
```

- Control logging from within code or through external configuration file

W08:L01: Cloning

- Making a faithful copy of an object is a tricky problem
- Java provides a clone() function in Object that does shallow copy
- However, shallow copy aliases nested objects
- Deep copy solves the problem, but inheritance can create complications
- To force programmers to consciously think about these subtleties, Java puts in some checks to using clone()
- Must implement marker interface Cloneable to allow clone()
- clone() is protected by default. override as public if needed
- clone() in Object throws CloneNotSupportedException, which must be taken into account when overriding

W08:L02: Type inference

- Automatic type inference can avoid redundancy in declarations

  `Employee e = new Employee(...)`

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Challenge is to do this statically, at compile-time
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value
- Be careful about format for numeric constants
- For classes, infer most constrained type
  - `e` is inferred to be `Manager`
  - `Manager` extends `Employee`
  - If `e` should be `Employee`, declare explicitly

```
var b = false; // boolean

var s = "Hello, world"; // String

var d = 2.0; // double

var f = 3.141f; // float

var e = new Manager(...); // Manager
```

W08:L03: Higher order functions

- Passing a function as an argument to another function
- In object-oriented programming, this is achieved using interfaces – encapsulate the function to be passed as an object
- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`
- Substitute wherever a functional interface is specified

```
String[] strarr = new ...;
Arrays.sort(strarr, (String s1, String s2) -> s1.length() - s2.length());
```

- Limited type inference is also possible
  - Java infers `s1` and `s2` are `String`

```
String[] strarr = new ...;
Arrays.sort(strarr, (s1, s2) -> s1.length() - s2.length());
```

- More complicated function body can be defined as a block

W08:L03: Higher order functions

- If the lambda expression consists of a single function call, we can pass that function by name – method reference
- We saw an example with adding entries to a Map object – here sum is a static method in Integer

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

- Here is the corresponding expression, assuming type inference

```
(i,j) -> Integer::sum(i,j)
```

- `ClassName::StaticMethod` – method reference is `C::f`, and corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> C::f(x1,x2,...,xk)
```

- `ClassName::InstanceMethod` – method reference is `C::f`, and called with respect to an object that becomes implicit parameter

```
(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)
```

- `object::InstanceMethod` – method reference is `o::f`, and arguments are passed to `o.f`

```
(x1,x2,..,xk) -> o.f(x1,x2,...,xk)
```

W08:L04: Streams

- We can view a collection as a stream of elements
- Process the stream rather than use an iterator
- Declarative way of computing over collections
- Create a stream, transform it, reduce it to a result
- Processing can be parallelized
  - `filter()` and `count()` in parallel
- Apply `stream()` to a collection
  - Part of Collections interface
- Use static method `Stream.of()` for arrays
- Create a stream, transform it, reduce it

```
long count = words.stream()
             .filter(w -> w.length() > 10)
             .count();
}

long count = words.parallelStream()
             .filter(w -> w.length() > 10)
             .count();
}
```

W08:L04: Streams (Cont.)

- Static method `Stream.generate()` generates a stream from a function
- `Stream.iterate()` — a stream of dependent values
- `filter()` to select elements – takes a predicate as argument
- `map()` applies a function to each element in the stream
- `flatMap()` flattens (collapses) nested list into a single stream
- Make a stream finite — `limit(n)`
- Skip n elements — `skip(n)`
- Stop when element matches a criterion — `takeWhile()`
- Start after element matches a criterion — `dropWhile()`
- Number of elements — `count()`
- Largest and smallest values seen - `max()` and `min()`
- First element — `findFirst()`
- What happens if the stream is empty? Return value is optional type

W09:L01: Optional Types

- `Optional<T>` is a clean way to encapsulate a value that may be absent
- Different ways to process values of type `Optional<T>`
  - Replace the missing value by a default
  - Ignore missing values
- `max()` and `min()` – what happens if the stream is empty?
- `max()` of empty stream is undefined – return value could be `Double` or `null`
- `Optional<T>` object – wrapper
- May contain an object of type `T` if value `present`, or no object
- Use `orElse()` to pass a default value

```
Optional<Double> maxrand =
    Stream.generate(Math::random)
    .limit(100)
    .filter(n -> n < 0.001)
    .max(Double::compareTo);
```

```
Double fixrand = maxrand.orElse(-1.0);
```

W09:L01: Optional Types (Cont.)

- Use `orElseGet()` to call a function to generate replacement for a missing value

```
Double fixrand = maxrand.orElseGet(
    () -> SomeFunctionToGenerateDouble
);
```

- Use `orElseThrow()` to generate an exception when a missing value is encountered

```
Double fixrand =
    maxrand.orElseThrow(
    IllegalStateException::new
);
```

- Use `ifPresent()` to test if a value is present, and process it

```
optionalValue.ifPresent(
    v -> Process v);
```

- Use `ifPresentOrElse` to Specify an alternative action if the value is not present

```
maxrand.ifPresentOrElse(
    v -> results.add(v),
    () -> System.out.println("No max")
);
```

W09:L01: Optional Types (Cont.)

- Creating an optional value
  - `Optional.of(v)` creates value v
  - `Optional.empty` creates empty optional
- Use `ofNullable()` to transform `null` automatically into an empty optional
- `map` applies function to value, if present; if input is `empty`, so is output

```
Optional<Double> maxrandsqr = maxrand.map(v -> v*v);
```

- Supply an alternative for a missing value using `or()`

```
Optional<Double> fixrand = maxrand.or(() -> Optional.of(-1.0));
```

- `flatMap` allows us to cascade functions with optional types
  - Use `flatMap` to regenerate a stream from optional values

W09:L02: Collecting results from streams

- Convert collections into sequences of values — streams
- `Stream` defines a standard iterator, use to loop through values in a stream
- Alternatively, use `forEach` with a suitable function

```
mystream.forEach(System.out::println);
```

- Can convert a stream into an array using `toArray()`

```
Object[] result = mystream.toArray();
String[] result = mystream.toArray(String[]::new);
```

- What if we want to convert the stream back into a collection? – use `collect()`
- Pass appropriate factory method from `Collectors`
- Create a list from a stream

```
List<String> result = mystream.collect(Collectors.toList());
```

- . . . or a set

```
Set<String> result = mystream.collect(Collectors.toSet());
```

- To create a concrete collection, provide a constructor

```
TreeSet<String> result =
    stream.collect(Collectors.toCollection(TreeSet::new));
```

W09:L02: Collecting results from streams (Cont.)

- `Collectors` has methods to aggregate summaries in a single object
  - `summarizingInt` works for a stream of integers

```
IntSummaryStatistics summary = mystream.collect(
        Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

- Methods to access relevant statistics – `getCount()`, `getMax()`, `getMin()`, `getSum()`, `getAverage()`,
- Similarly, `summarizingLong()` and `summarizingDouble()` return `LongSummaryStatistics` and `DoubleSummaryStatistics`
- Convert a stream of `Person` to a map – For `Person p`, `p.getID()` is key and `p.getName()` is value

```
Stream<Person> people = ...;
Map<Integer, String> nameToID = people.collect(
    Collectors.toMap(Person::getId, Person::getName)
);
```

W09:L02: Collecting results from streams (Cont.)

- Collect all ids with the same name in a list

```
Stream<Person> people = ...;
Map<String, List<Person>> nameTopersons = people.collect(
        Collectors.groupingBy(Person::getName)
);
```

- Instead, may want to partition the stream using a predicate – Partition names into those that start with A and the rest

```
Stream<Person> people = ...;
Map<Boolean, List<Person>> aAndOtherPersons =
    people.collect(
        Collectors.partitioningBy(
            p -> p.getName().substr(0,1).equals("A")
        )
);
```

W09:L03: Input/output streams

- Input: read a sequence of bytes from some source – a file, an internet connection, memory
- Output: write a sequence of bytes to some source – a file, an internet connection, memory
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`
- Close a stream when done — release resources
- Flush an output stream — output is buffered
- Similarly, write one or more bytes using – `OutputStream`
- Create an input stream attached to a file – `FileInputStream`
- Create an output stream attached to a file – `FileOutputStream`
- Overwrite or append? – Pass a boolean second argument to the constructor
- `Scanner` class – apply to any input stream – many read methods
- To write text, use `PrintWriter` class – apply to any output stream

W09:L03: Input/output streams (Cont.)

- To read binary data, use `DataInputStream` class – many read methods
- To write binary data, use `DataOutputStream` class
- Buffering an input stream – reads blocks of data – `BufferedInputStream()`
- Similarly, write blocks using – `BufferedOutputStream()`
- `PushBackStream` can only `read()` and `unread()`
- Java has a whole zoo of streams for different tasks – random access files, zipped data, . . .

W09:L04: Serialization

- `DataInputStream` and `DataOutputStream` – read and write low level units like bytes, integers, floats, characters, . . .
- Can we export and import objects directly?
  - Backup objects onto disk, with state
  - Restore objects from disk
  - Send objects across a network
- Serialization and deserialization
- To write objects, Java has another output stream type, `ObjectOutputStream`
- Use `writeObject()` to write out an object
- To read back objects, use `ObjectInputStream`
- Retrieve objects in the same order they were written, using `readObject()`
- Class has to allow serialization — implement marker interface `Serializable`
- Some objects should not be serialized – mark such fields as `transient`
- Can override `writeObject()`
  - `defaultWriteObject()` writes out the object with all non-transient fields
  - Then explicitly write relevant details of transient fields

W10:L01: Concurrent Programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Logically parallel actions within a single application
  - Operated on same local variables
  - Communicate via “shared memory”
  - Context switches are easier

W10:L01: Creating threads

- Have a class extend `Thread`
- Define a function `run( )` where execution can begin in parallel
- Invoke `start( )`, initiates `run( )` in a separate thread
- Directly calling `run( )` does not execute in separate thread!
- Cannot always extend Thread, Instead, implement `Runnable( )`
- To use class, `Runnable( )` explicitly create a `Thread` and `strat( )` it

W10:L02: Race conditions and mutual exclusion

- Race condition: Concurrent update of a shared variable can lead to data inconsistency
- Critical sections: sections of code where shared variables are updated
- Mutual exclusion: at most one thread at a time can be in a critical section

W10:L03:Mutual Exclusion

- At most one thread at a time can be in a critical section
- Using a two-(int)valued variable
  - Leads to starvation
- Using two boolean variables
  - Leads to deadlock
- Petersen Algorithm - Combination of a two-valued variable and two boolean variables
  - Avoids both starvation and deadlock
  - But difficult to generalize to more than two threads
- Bakery Algorithm - The thread itself acquires a token number by incrementing the current token number by 1.
  - If two threads get the same number, then some system parameter is used to break the tie.
  - To avoid this, the checking for max and incrementing should be made an atomic step.

W10:L03:Mutual Exclusion

- Using a two-valued variable

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

W10:L03:Mutual Exclusion

- Using two boolean variables

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

W10:L03:Mutual Exclusion - Peterson Algorithm

- Using a single two-valued variable and two boolean variables

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

W10:L04:Test-and-set

- Check the current value of a variable, then update it
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive

W10:L04:Semaphores

- Programming language primitive for `test-and-set` is given by **Semaphores**.
-

```
Thread 1
...
P(S);
// Enter critical section
   ...
// Leave critical section
V(S);
...
```

```
Thread 2
...
P(S);
// Enter critical section
   ...
// Leave critical section
V(S);
...
```

- Semaphores guarantee
  - Mutual exclusion
  - Freedom from starvation
  - Freedom from deadlock

W11:L1: Threads in JAVA

- Threads can be created by extending **Thread** class or implementing **Runnable** interface.
- We define a function **run** which is executed by the spawned threads of that class.
- Invoking the **start** method of the thread object initiates the `run` method on a separate thread (not the `main` thread).
- **Thread.sleep(arg)** suspends the thread for `arg` milliseconds.
  A thread does not release its lock (if already gained) while in sleep.
- **join()** method makes the thread executing this instruction wait for the thread on which join is called to complete its execution.
- Both `join` and `sleep` methods can throw **interruptedException**.
- To check a thread's interrupt status one can use `t.isInterrupted()`. It does not clear the flag
- While writing a multithreaded program, one can either define multiple classes with their own `run` methods working on different features of the code or different threads can be filtered by their name inside one `run` method and corresponding utility functions are called using these threads.

W11: Thread: Synchronization

- Shared variables between threads can show inconsistent values as a result of context switching of threads arbitrarily (race condition). In order to attain atomicity while updating shared variables the **synchronized** keyword is used.
- JAVA allows a method or a block of code to be executed in synchronized manner. In case of synchronization block, we mention the object whose lock would be held by the thread executing the synchronized block.
- Programs in which the critical section is synchronized will maintain consistent value of the shared variables irrespective of the degree of concurrency (number of threads).
- Intercommunication between threads can be done by using `wait()` and `notify()/notifyAll()` methods.
  - **wait()**: is used to suspend a thread. The thread loses its lock and goes to `BLOCKED` state.
  - **notify()**: signals one arbitrary thread waiting on that object that the object lock is now free to be gained.
    notifyAll(): signals all waiting threads, but one of them will be picked by the system to be executed.
- **wait, notify, notifyAll** must always be used inside synchronized blocks/methods otherwise **IllegalMonitorStateException** is thrown.

W11: Thread State

- Life Cycle of a thread in JAVA:
  - **New:**
    Created but, start() is not invoked.
  - **Runnable:**
    start() method has been invoked and the thread is ready to be scheduled.
  - **Running:**
    the current thread is under execution.
  - **Blocked:**
    the thread is suspended by wait() method, it has lost its lock.
  - **Timed Waiting:**
    the thread is sleeping, holding its lock
  - **Terminated:**
    Either the thread has finished its job and has terminated normally or it has terminated because of some abnormal event like segmentation fault/unhandled exception.

W11: Re-entrant Locks and Threadsafe Collections

- JAVA provides semaphore like mechanism to implement concurrency in programs using reentrant locks.
  `ReentrantLock` class
- A reentrant lock object has methods `lock()` and `unlock()` just like `P(s)` and `V(s)`.
- Always put the unlock() under a `finally` block.
- Why re entrant?
  - Thread already holding lock of an object can reacquire it.
    very useful while working with recursive codes which must be executed atomically but in a multithreaded environment.
- Threadsafe collections guarantees consistency of individual updates/accesses done by multiple threads on the same collection.
- Sequence of updates is still not guaranteed in threadsafe collections.
- JAVA provides built in collections which are threadsafe
  - ConcurrentHashMap
  - BlockingQueue
- A threadsafe collection will never throw `ConcurrentModificationException`

W12:GUI programming

- How to interact with the Java program?
  - Using CLI(Command Line Interface).
  - Using GUI(Graphical User Interface).
- How to create GUI?
  - Using AWT(Abstract Window Toolkit).
  - Using Swing.
- Both AWT and Swing are used to create GUI.
  - AWT components heavyweight.
  - Swing components lightweight.

Steps to create GUI

- create a class by extending JFrame/Frame.
- Create GUI components.
- Add GUI components to the panel.
- Add panel to the frame.
- call setVisible(true) to visible the GUI screen.

Event Driven Programming

- AWT and Swing just create GUI.
- If we want to perform operations behind the GUI screens, we should use event driven programming.
- Button
  - ActionListener
  - ActionEvent
  - public abstract void actionPerformed(ActionEvent e)
- KeyBoard
  - KeyListener
  - KeyEvent
  - public abstract void keyTyped(KeyEvent e);
  - public abstract void keyPressed(KeyEvent e);
  - public abstract void keyReleased(KeyEvent e);