IITM JAVA

Week 1

- Imperative vs Declarative programming

```
# Imperative structure
def sumlist(l):
        my_sum = 0
        for x in l:
                my_sum =+ x
        return my_sum

# Declarative structure
def sumlist(l):
        if l == []:
                return 0
        else:
                return (l[0] + sumlist(l[1:]))
```

```
# Imperative structure
def sumsquareeven(n):
        my_sum = 0
        for c in range(n+1):
                if x%2 == 0:
                        my_sum =+ x**2
        return my_sum

# Declarative structure
def even(x):
        return (x%2 == 0)
def square(x):
        return x**2
def sumsquareeven(n):
        return sum(map(square, filter(even, range(n+1))))
```

- Declarative style avoids storing variables
- **Types**
  - interpreting data stored in binary in a consistent manner
    - nature and range of allowed values
    - operations that are permitted on these values
  - naming concepts and structuring our computation
  - catching bugs early
  - Dynamic vs static typing
    - each variable we use has a type
    - Dynamic typing: Python determines the type based on the current value
      - difficult to catch errors
      - no type if there's no value
    - Static typing: associate a type in advance with a name
      - empty user defined objects
      - even simple type "synonyms" can help clarify code
      - more elaborate types – abstract datatypes and object-oriented programming
  - Static analysis
    - identify errors as early as possible – saves cost, effort

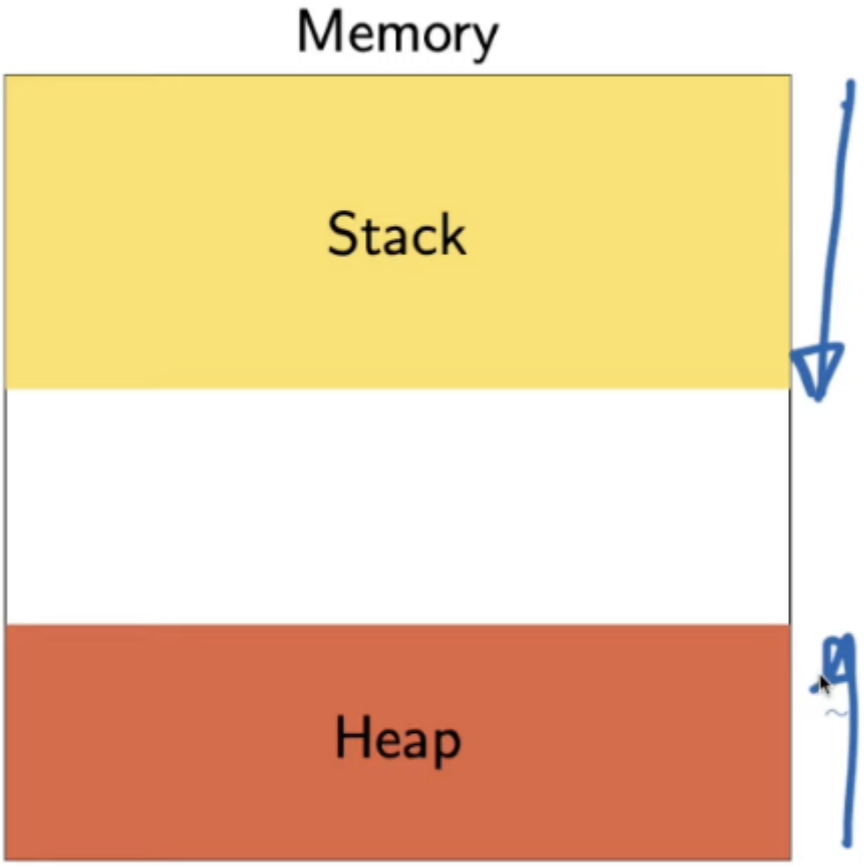
    - compilers cannot check that a program will work correctly
    - compilers can detect type errors at compile-time
    - Compilers can also perform optimisations based on static analysis
        - reorder statement to optimise reads and writes
        - store previously computed expressions to re-use later

- ## Memory Management
    - variables store intermediate values during computation
    - Scope of a variable
        - when the variable is available for use
    - Lifetime of a variable
        - how long the storage remains allocated
        - can exceed the scope of a variable
    - Memory stack
        - each function needs storage for local variables
        - Create activation record when function is called
        - Activated records are stacked: popped when function exits
    - Two ways to initialise parameters
        - Call by value — copy the value
            - updating the value inside the function has no side-effect
        - Call by reference — parameter points to same location as argument
            - can have side-effects

    - **Heap**

        - Function that inserts a value in a linked list
            - storage for new node allocated inside function
            - node should persist after function exits
        - Separate storage for persistent data
            - dynamically allocated vs statically declared
            - not the same as the heap data structure

- Heap storage outlives activation record



- Managing heap storage
  - on the stack, variables are deallocated when a function exits
  - how do we 'return' unused storage on the heap?
    - after deleting a node in linked list, deleted nodes are now dead storage
  - manual memory management
    - programmer explicitly requests and returns heap storage
    - error-prone — memory leaks, invalid assignment
  - automatic garbage collection (Java, Python, ...)
    - run-time environment checks and cleans up dead storage
    - convenient for programmer vs performance penalty

Abstraction and Modularity

- Stepwise refinement
  - Begin with a high level description of the task
  - Refine the task into subtasks
  - Further elaborate each subtask
  - Subtasks can be coded by different people
- Program refinement — focus on code, nor much change in data structures
- Data refinement
  - Banking application for example
    - typical functions: `CreateAccount()`, `Deposit()/Withdrawal()`, `PrintStatement()`
  - Refine `PrintStatement()` to include `PrintTransactions()`
- Modular software development
  - use refinement to divide solution into components
  - components are described in terms of
    - Interfaces — what is visible to other components, typically function calls
    - Specification — behaviour of the component, as visible through interface
- Data abstraction

    - Abstract data types (ADTs)
    - Set of values along with operations permitted on them
    - Internal representation should not be accessible
    - Interaction restricted to public interface
  - Object-oriented programming
    - ADTs in a hierarchy
    - Implicit reuse of implementation — subtyping, inheritance

Object Oriented Programming

- Uniform way of encapsulating different combinations of data and functionality
- Distinguishing feature of OOP
  - Abstraction
  - Subtyping
  - Dynamic Lookup
  - Inheritance
- Abstraction
  - Objects are similar to abstract datatypes
    - public interface
    - private implementation
    - changing the implementation shouldn't affect interactions with the object
  - Data-centric view of programming
    - Focus on what data we need to maintain and manipulate
- Subtyping
  - Arrange types in a hierarchy
    - a subtype is a specialisation of a type
    - if A is a subtype of B, wherever an object of B is needed, an object of type A can be used
- Dynamic lookup
  - Whether a method can be invoked on an object is a static property — type-checking
  - How the method acts is a dynamic property of how the object is implemented
  - Different from overloading
    - Internal implementation is different, but choice is determined by static type
    - overloading happens when we apply different functions with the same name to different types
  - Dynamic lookup
    - A variable `v` of type `B` can refer to an object subtype `A`
    - Static type of `v` is `B`, but the method implementation depends on run-time type A
- Inheritance
  - Re-use of implementations
  - Example: different types of employees
    - `Employee` objects store basic personal data, date of joining, etc.
    - `Manager` objects can add functionality to the `Employee` class
  - `A` can inherit from `B` iff `A` is a subtype of `B`
    - Subtyping is a relationship of interfaces
    - Inheritance is a relationship of implementations
- Subtyping vs inheritance
  - A deque is a double-ended queue
  - We can implement a stack or a queue using a deque
    - Stack: use only `insert-front(), delete-front()`
    - Queue: use only `insert-rear(), delete-front()`
  - Stack and Queue inherit from Deque, but they're not subtypes of Deque
    - If `v` of type Deque points to an object of type `Stack`, cannot invoke `insert-rear(), delete-rear()`
    - subtypes need to allow all the functions, but that's not the case in inheritance

Classes and Objects

- Class

Week 2

First Taste of Java

- printing "hello, world" in Java

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

- All code in Java lives within a class
  - no free floating function, unlike Python
  - modifier `public` specifies visibility
- How the program starts
  - Fix a function name that will be called by default
  - From C, the convention is to call this function `main()`, this is where all execution starts from
- Explanation of `main()` function in the code above
  - Input parameter is an array of string; command line arguments
  - No output, so return type is `void`
- Visibility
  - Function has to be available to run from outside the class
  - Modifier `public`
- Availability
  - Functions defined inside classes are attached to objects
  - How can we create an object before starting?
  - Modifier `static` — function that exists independent of dynamic creation of objects
  - `static` says that the function can be invoked without having to create an object of that class
- `.out` is a stream object sefine in `System`
  - like a file handle
  - note that `out` must also be `static`
- `println()` is a method associated with streams
  - prints argument with a newline, like Python `print()`
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `hellowordld` in the file `helloworld.java`
- Java programs interpreted on Java Virtual Machine (JVM)
  - JVM provides a uniform execution environment across operating systems
  - Semantics of Java is defined in terms of JVM, OS-independent
  - "Write once, run anywhere"
- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates a bytecode file `helloworld.class`
  - `java helloworld` interprets and runs bytecode in `helloworld.class`
- Note:
  - javac requires file extension `.java`
  - java should not be provided file extension `.class`
  - javac automatically follows dependencies and compiles all classes required
    - sufficient to trigger compilation for class containing `main()`

Basic Datatypes in Java

- Scalar types
  - int, long (larger version of integer), short, byte (takes exactly one byte of memory)
  - float (fractional values), double (larger range of exponents)
  - char

- boolean

Type	Size in bytes
int	4
long	8
short	2
byte	1
float	4
double	8
char	2
boolean	1

  - sizes of each scalar type
  - Characters are written with single-quotes (only); double quotes denote strings
  - Boolean constants are `true`, `false`
  - `float pi = 3.14159f` : Append `f` after number for `float`, else Java will interpret it as `double`
  - `final float pi = 3.14159f` : adding `final` means that the variable value cannot be further modified
- Operators, shortcuts, type casting
  - Arithmetic operators: +, -, *, /, %
  - no separate integer division operator
  - `float f = 22/7` : the value assigned to `f` will be `3.0` because Java treats the division of two integers as integer division
  - `Math.pow(a,n)` returns a^n
  - `a++` same as `a = a+1` ; `a--` same as `a = a-1`
  - `a += 5` same as `a = a+5`
  - `a *= 5` same as `a = a*5`
- Strings
  - `String s = "Hello", t = "world";`
  - `String u = s + " " + t`
  - Strings are not arrays of characters
  - `s = s.substring(0,3) + "p!";`
- Arrays
  - all elements should be of the same type
  - Typical declaration: `int[] a; a = new int[100]`
    - Combined: `int[] a = new int[100]`
  - `a.length` gives size of `a`
  - Arrays can also be defined with constants

```
int[] a;
int n;

n = 10;
a = new int[n];

n = 20;
a = new int[n];
```

```
a = {2,3,5,7,11};
```

- ## Control Flow in Java
  - Conditional execution `if (condition) {...} else {...}`
    - `else` is optional
    - no `elif` like in Python
  - Conditional loops
    - `while (condition) {...}`
    - `do {...} while (condition)`
  - Iteration: two kinds of `for`
    - `for (init; cond; upd) {...}`
      - `init` is initialisation
      - `cond` is terminating condition
      - `upd` is update
      - example: `for(int i=0; i < n; i++) {...}`
    - `for (int x: a) { sum += x; }` where `a` is a sequence (array, etc.)
  - Multiway branching statement `switch`
    - `switch` selects between different options

```
public static void printsign(int v){
        switch (v) {
                case -1: {
                        System.out.println("Negative");
                        break;
                }
                case 1: {
                        System.out.println("Positive");
                        break;
                }
                case 0: {
                        System.out.println("Zero");
                        break;
                }
        }
}
```

    - Options have to be constants; cannot use conditional expressions

- ## Defining Classes and Object in Java
  - definition block using `class`, with class name

```
public class Date {
        private int day, month, year;

        public void setDate(int d, int m, int y){
                this.day = d;
                this.month = m;
                this.year = y;
        }

        public int getDay(){
                return(day);
        }

        public int getMonth(){
                return(month);
        }

        public int getYear(){
```

```
                return(year);
        }
}
```

- modifier `public` to indicate visibility
- Java allows `public` to be omitted, it won't be wrong syntactically if you don't mention `public`
- Default visibility is public within the current `package`

- Instance variables
  - each concrete object of type `Date` will have local copies of `date, month, year`, marked `private`
  - can also have `public` instance variables, but breaks encapsulation
- creating an object

```
public void UseDate() {
        Date d;
        d = new Date();
        ...
}
```

  - `new` creates a new object
- Initialise objects

```
public class Date {
        private int day, month, year;

        // Constructor function like __init__ in python
        public Date(int d, int m, int y){
                day = d;
                month = m;
                year = y;
        }

        // Constructor with hardcoded element
        public Date(int d, int m){
                day = d;
                month = m;
                year = 2020;
        }

        // Constructor that takes argument of another Date object
        public Date(Date d){
                this.day = d.day;
                this.month = d.month;
                this.year = d.year;
        }
}

public void UseDate() {
        Date d1, d2;
        d1 = new Date(13,8,2020);
        d1 = new Date(13,8); // will also work
        d2 = new Date(d1);
}
```

Basic Input and Output in Java

- to print data: `System.out.println("hello, world")`
- Input using `Console` class

```
Console cons = System.console();
String username = cons.readLine("User name: ");
```

```
char[] passwd = cons.readPassword("Password: ");
```

- Input using `Scanner` class

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

Week 3

- Relationship between classes
  - Dependence
    - `Order` needs `Account` to check credit status
    - Robust design minimises dependencies, or coupling between classes
  - Aggregation
    - `Order` contains `Items` objects
  - Inheritance
    - One object is an enhanced version of another
    - `ExpressOrder` inherits from `Order`
    - Extra methods to compute shipping charges, priority handling
- ## Subclasses
  - An `Employee` class

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

        // Constructors
        public Employee(String n, double s){
                name = n; salary = s;
        }
        public Employee(String n){
                this(n,500.00);
        }

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

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

        // other methods
        public double bonus(float percent){
                return (percent/100.0) * salary;
        }
}
```

  - Managers are special types of employees with extra features; `Manager` subclass

```
public class Manager extends Employee{
        private String secretary;
        public boolean setSecretary(name s){ ... }
        public String getSecretary(){ ... }
```

```
        // Constructor for Manager
        public Manager(String n, double s, String sn){
                super(n,s); // Super calls Employee constructor
                secretary = sn;
        }

        // Uses parent class bonus() via super
        // Overrides definition in parent class
        double bonus(float percent){
                return 1.5 * super.bonus(percent);
        }
}
```

    - Managers are a subset of Employees
- `Manager` objects do not automatically have access to private data of parent class. Need to use `super` for that.

- ## Inheritance
    - Subclass has more features than parent class
        - Subclass *inherits* instance variables, methods from parent class
    - Every `Manager` is an `Employee`, but not vice versa
    - Can use a subclass in place of a superclass
        - `Employee e = new Manager(...)`
    - But this will not work: `Manager m = new Employee(...)`
    - Can initialise an array of managers:
        - `Employee[] e = new Manager(...)[100]`
- ## Dynamic dispatch
    - `Employee e = new Manager(...)`
        - if we call `e.bonus(p)` which bonus function do we use?
            - Static: Use `Employee.bonus()`
            - Dynamic: Use `Manager.bonus()` :: This is the function that'll be used
    - Dynamic dispatch turns out to be more useful: default in Java, optional in languages like C++ (virtual function)
- ## Polymorphism
    - Every `Employee` in `emparray` "knows" hot to calculate its bonus correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager m = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i=0; i < emparray.length; i++){
        System.out.println(emparray[i].bonus(5.0))
}
```

    - a.k.a. *runtime polymorphism* or *inheritance polymorphism*
- ## Functions, signatures and overloading
    - Signature of a function: its name and list of arguments
    - Can have different function with the same name and different signatures
    - Java in-built class `Arrays` has a method `sort` to sort arbitrary scalar arrays. Made possible by overloaded methods defined in class `Arrays`

```
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[]
  ...
}
```

- 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

- **Type casting**
  - Consider `Employee e = new Manager(...)`. Can we get `e.setSecretary()` to work?
    - Static type-checking disallows this
    - Type casting — convert `e` to `Manager`: `((Manager) e).setSecretary(s)`
  - Can test if `e` is a `Manager`:
    - `if (e instanceof Manager){ ((Manager) e).setSecretary(s) }`
  - Can also use type casting for basic types
    - `double d = 29.98; int nd = (int) d;`
- **Java class hierarchy**
  - multiple inheritance is not allowed in Java
  - universal superclass `Object`
  - For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`
  - Can exploit tree structure to write generic functions
    - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
        int i;
        for (i=0; i<objarr.length(); i++){
                if (objarr[i] == 0) {return i};
        }
```

```
            return (-1);
    }
```

- operator `==` is pointing to a function (`equals()`) in the Java superclass `Object` which checks if both the inputs are of the same object type
- If, for example, we want to override the `equals()` function such that it compares the state, not the object type:

```
public boolean equals( Object d ){
        if (d instanceof Date){
                Date myd = (Date) d;
                return ((this.day == myd.day) && (this.month == myd.month) && (this.year
== myd.year))
        }
        return false;
}
```

- Overriding functions
  - Overriding looks for "closest" match
  - Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
  - Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

  - `public boolean equals(Manager m)` is compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`
    - `boolean equals (Employee e)` will be called because it's the closest match

Subtyping vs Inheritance

- Subtyping
  - Capabilities of 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
- 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()`
- Example:
  - `queue` methods: `insert-rear`, `delete-front`
  - `stack` methods: `insert-front`, `delete-front`
  - `deque` methods: `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
  - Subtyping:
    - `deque` has more functionality than `queue` or `stack`
    - `deque` is subtype of both these types
  - Inheritance
    - Can suppress two functions in a `deque` and use it as a `queue` or `stack`
    - Both `queue` and `stack` inherit from `deque`

Java modifiers

- Modifiers
  - `public` vs `private` to support encapsulation of data
  - `static` for entities defined inside classes that exist without creating objects of the class

    - `final` for values that cannot be changed
    - can be applied to classes, instance variable and methods
- public vs private
    - Typically, instance variables are `private`
    - Methods to query (accessor) and update (mutator) the state are `public`
- private methods
    - can't be accessed outside the class
    - useful for internal calculations
- static components
    - for components that exist without creating objects
    - Useful constants like `Math.PI`
    - are `public`
    - `private static` components
        - for unique identifiers required inside the class

```
public class Order {
        private static int lastorderid = 0; // independent of the objects
        private int orderid;
        ...
        public Order(...){
                lastorderid++;
                orderid = lastorderid;
                ...
        }
}
```

- final components
    - cannot be updated

Week 4

- **Abstract Classes**
    - If there is a functionality that is needed in all the subclasses, but with modifications in each subclass. for example: `Shape` is the main class, and circle, square, rectangle, etc. are subclasses and the functionality needed is `perimeter()`
    - Provide an abstract definition in `Shape` : `public abstract double perimeter()`
    - Works like a template that is being written in the main class. `Shape` must itself be declared `abstract`, i.e. the moment you have at least one of the methods defined in that class as `abstract` then the entire class needs to be declared `abstract`

```
public abstract class Shape{
        ...
        public abstract double perimeter();
        ...
}
```

    - can declare variables whose type is an abstract class

```
Shape shapearr[] = new Shape[3];
int sizearr[] = new int[3];

shapearr[0] = new Circle(...);
shapearr[1] = new Square(...);
shapearr[2] = new Rectangle(...);
```

```
for (i=0; i<2; i++){
        sizearr[i] = shapearr[i].perimeter();
        ... // each shapearr[i] calls the appropriate method
}
```

- to specify generic properties; to use it for sorting, for example

```
public abstract class Comparable{
        public abstract int cmp(Comparable s);
                // return -1 if this < s,
                //         0 if this == s,
                //        -1 if this > s
}

public class SortFunctions{
        public static void quicksort(Comparable[] a){
                ...
                // Usual code for quicksort, except to compare
                // a[i] and a[j] we use a[i].cmp(a[j])
        }
}

// To use this definition of quicksort, we write:
public class Myclass extends Comparable{
        private double size; // qty used for comparison

        public int cmp(Comparable s){
                if (s instanceof Myclass){
                        // compare this.size and ((Myclass) s).size
                        // Note the cast to access s.size
                }
        }
}
```

- **Multiple inheritance**
  - Can we sort `Circle` using generic functions in `SortFunctions`?
    - `Circle` extends `Shape`
    - Java does not allow `Circle` to also extend `Comparable`
  - An interface is an abstract class with no concrete components i.e., all the methods/components are `abstract`

```
public interface Comparable{
        public abstract int cmp(Comparable s);
}
```

  - Interface allows you to avoid contradictory implementations in case there are two different implementations among the multiple inherited classes.
  - Class that extends an interface is said to `implement` it

```
public class Circle extends Shape implements Comparable{
        public double perimeter(){...};
        public int cmp(Comparable s){...};
}
```

  - Can extend only one class, but can implement multiple interfaces

- **Interfaces**
  - All methods must be abstract
  - A class `implements` an interface

- Exposing limited capabilities
  - Generic `quicksort` for any datatype that supports comparison
  - Express this capability by making the argument type `Comparable[]`. The only information that `quicksort` needs about the underlying type, all other aspects are irrelevant
- Adding methods to interfaces
  - Can define static functions in interfaces; cannot access instance variables; invoked directly or using interface name: `Comparable.cmpdoc()`

```
public interface Comparable{
        public static String cmpdoc(){
                String s;
                s = "Return -1 if this < s, ";
                s = s + "0 if this == s, ";
                s = s + "+1 if this > s.";
                return (s);
        }
}
```

  - Can define default functions; class can override these; invoke like normal method, using object name: `a[i].cmp(a[j])`

```
public interface Comparable{
        public default int cmp(Comparable s) {
                return 0;
        }
}
```

- Dealing with conflicts
  - Conflict between two interfaces: Subclass **must** provide a fresh implementation

```
public interface Person{
        public default String getName() {
                return ("No name");
        }
}

public interface Designation{
        public default String getName() {
                return ("No designation");
        }
}


// CONFLICT BETWEEN TWO INTERFACES
public class Employee implements Person, Designation{
        ...
        public String getName(){
                ....
        }
}
```

  - Conflict between a class and an interface: the method inherited from the class **wins**

```
public class Person{
        public String getName() {
                return ("No name");
        }
}

public interface Designation{
```

```
        public default String getName() {
                return ("No designation");
        }
}


// CONFLICT BETWEEN TWO INTERFACES
public class Employee extends Person implements Designation{
        ...
        public String getName(){
                ....
        }
}
```

- **Private Classes**
  - Implementation of a LinkedList

```
public class Node {
        public Object data;
        public Node next;
        ...
}

public class LinkedList {
        private int size;
        private Node first;

        public Object head() {
                Object returnval = null;
                if (first != null){
                        returnval = first.data;
                        first = first.next;
                }
                return (returnval);
        }
}
```

  - Implementation using `Node` class as a private class

```
public class LinkedList{
        private int size;
        private Node first;

        public Object head() {...}

        public void insert(Object newdata){
        ...
        }
        private class Node {
                public Object data;
                public Node next;
                ....
        }
}
```

  - Objects of private class can see private components of enclosing class
- **Controlled Interaction with Objects**
  - Example of a common implementation of a class

```
public class Date {
        private int day, month, year;
```

```
        public void getDay(int d) {...}
        public void getMonth(int m) {...}
        public void getYear(int y) {...}

        public void setDate (int d, int m, int y) {
                ...
                // Validate d-m-y combination
        }
}
```

- Querying a database
  - Interaction with state: whether or not a method is allowed depends on the status of the user. For example: is the user logged-in or not
  - To maintain any state information is by creating an object
  - Example implementation: the `QIF` interface is getting returned, which is public but the actual object being returned is private i.e., `QueryObject`

```
public interface QIF{
        public abstract int getStatus(int trainno, Date d);
}


public class RailwayBooking {
        private BookingDB railwaydb;

        public QIF login(String u, String p) {
                QueryObject qobj;
                if (valid_login(u,p)) {
                        qobj = new QueryObject();
                        return qobj;
                }
        }

        private class QueryObject implements QIF {
                public int getStatus (int trainno, Date d) {
                        // Return number of seats available
                        // on train number trainno on date d
                        ...
                }
        }
}
```

  - The above implementation allows unlimited number of queries so the user can pass the query object to another program. To restrict that, allow only a finite life of `QueryObject` by maintaining a counter inside the object:

```
private class QueryObject implements QIF {
        private int numqueries;
        private final int QLIM;

        public int getStatus (int trainno, Date d) {
                if (numqueries < QLIM) {
                        // respond, increment numqueries
                }
        }
}
```

- **Callbacks**
  - `Myclass m` creates a `Timer t`
  - Start `t` to run in parallel (`Myclass m` continues to run)

- `Timer t` notifies `Myclass m` when the time limit expires
- Code

```
public class Myclass {
        public void f() {
                ...
                Timer t = new Timer(this);
                // this object created t
                ...
                t.start();
                ...
        }

        public void timerdone() {...}
}

public class Timer implements Runnable {
        // interface 'Runnable' means Timer can run in parallel

        private Myclass owner;

        public Timer(Myclass o) {
                owner = o; // My creator
        }
        public void start() {
                ...
                owner.timerdone(); // I'm done
        }
}
```

- The above code `Timer` only works with `Myclass`. To make it generic:

```
public class Timer implements Runnable {
        private Object owner;

        public Timer (Object o) {
                owner = o;
        }

        public void start() {
                ...
                ((Myclass) owner).timerdone();
        }
}
```

- Implementing using interfaces

```
public interface Timerowner {
        public abstract void timerdone();
}

public Myclass implements Timerowner {
        public void f() {
                ...
                Timer t = new Timer(this);
                ...
                t.start();
        }

        public void timerdone() {...}
}

public class Timer implements Runnable {
```

```
        private Timerowner owner;

        public Timer(Timerowner o) {
                owner = o;
        }

        public void start() {
                ...
                owner.timerdone();
        }
}
```

- **Iterators**
  - Encapsulate the functionality of an iterator in an interface called `Iterator`

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

  - Create an `Iterator` object and export it

```
public class Linearlist {

        private class Iter implements Iterator {
                private Node position;
                public Iter() {...} // Constructor
                public boolean has_next() {...}
                public Object get_next() {...}
        }

        // Export a fresh Iterator
        public Iterator get_iterator() {
                Iter it = new Iter();
                return it;
        }
}
```

Week 5

- **Polymorphism Revisited**
  - polymorphism: effect of dynamic dispatch
    - `S` is a subclass of `T`
    - `S` overrides a method `f()` defined in `T`
    - Variable `v` of type `T` is assigned to an object of type `S`
    - `v.f()` uses the definition of `f()` from `S` rather than `T`
  - Structural polymorphism
    - Use the Java class hierarchy
    - Polymorphic `reverse`

```
public void reverse (Object [] objarr) {
        Object tempobj;
        int n = objarr.length;
        for (i = 0; i < n/2; i++) {
                tempobj = objarr[i];
                objarr[i] = objarr[(n-1)-i];
```

```
                objarr[(n-1)-i] = tempobj;
        }
}
```

- Polymorphic `find`

```
public int find (Object [] objarr, Object o) {
        int i;
        for (i = 0; i < objarr.length; i++) {
                if (objarr[i] == o) {return i};
        }
        return (-1);
}
```

  - `==` translates to `Object.equals()`
- Polymorphic `sort`

```
public interface Comparable {
        public abstract int cmp(Comparable s);
}

public class SortFunctions {
        public static void quicksort(Comparable [] a) {
                ...
                // Usual code for quicksort, except that
                // to compare a[i] and a[j] we use a[i].cmp(a[j])
        }
}
```

- Polymorphic `copy`

```
public static void arraycopy(Object[] src, Object[] tgt) {
        int i, limit;
        limit = Math.min(src.length, tgt.length);
        for (i=0; i<limit; i++) {
                tgt[i] = src[i];
        }
}
Datep[] datearr = new Date[10];
Employee[] emparr = new Employee[10];
arraycopy( datearr , emparr); // Run-time error

public class Ticket {...}
public class Eticket extends Ticket {...}
Ticket[] tktarr = new Ticket[10];
Eticket[] etktarr = new Eticket[10];
arraycopy( etktarr , tktarr); // Allowed
arraycopy( tktarr , etktarr); // Run-time error
```

- **Generic Programming**
  - Polymorphic `reverse` with type quantifier

```
public <T> void reverse (T[] objarr) {
        T tempobj;
        int len = objarr.length;
        for (i=0; i<n/2; i++) {
                tempobj = objarr[i];
                objarr[i] = objarr[(n-1)-i];
                objarr[(n-1)-i] = tempobj;
```

```
        }
}
```

- Polymorphic `copy` with type quantifier

```
public <S extends T, T> static void arraycopy (S[] src, T[] tgt) {
        int i, limit;
        limit = Math.min(src.length, tgt.length);
        for (i=0; i<limit; i++) {
                tgt[i] = src[i];
        }
}
```

- Instantiate generic classes using concrete type

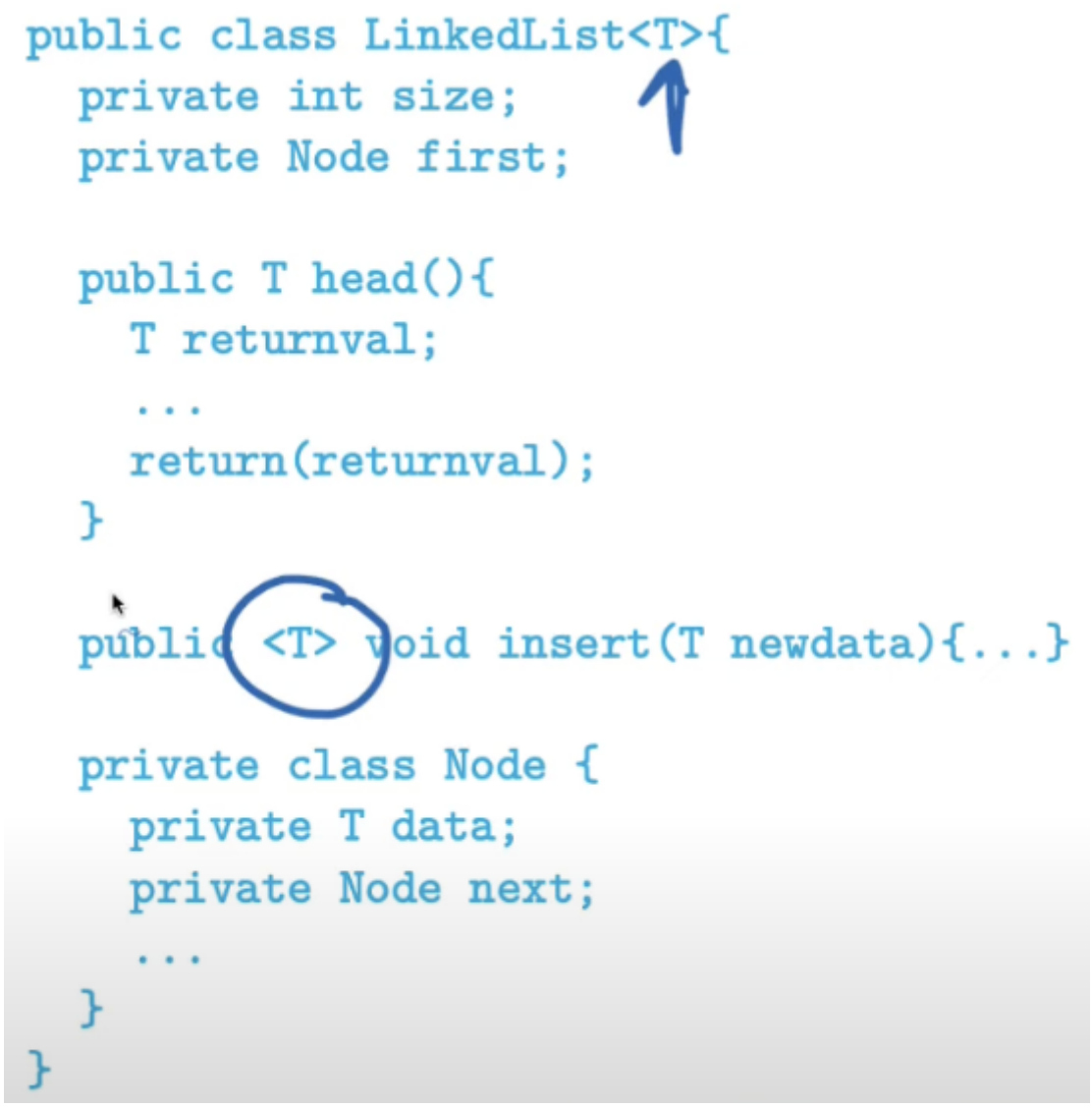
```
public class LinkedList <T> {...}

LinkedList<Ticket> ticketlist = new LinkedList<Ticket>();
LinkedList<Date> datelist = new LinkedList<Date>();

Ticket t = new Ticket();
Date d = new Date();

ticketlist.insert(t);
ticketlist.insert(d);
```

- If type variables are already defined a the 'class level', redefining them at the 'methods level' throws an error unless the method is `static`



- **Generics and Subtyping**
  - If `S extends T` then `S[] extends T[]`
  - We can make the method generic by introducing a type variable. If the variable type is not actually used inside the function, `?` can be used as a wildcard type variable

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

  - Can define variables of a wildcard type

```
public class LinkedList<T>{...}
```

```
LinkedList<?> l;
```

- But need to be careful about assigning values

```
public class LinkedList<T>{...}
LinkedList<?> l = new LinkedList<String>();
l.add(new Object()); // Compile time error
```

- Bounded wildcards
  - Suppose `Circle`, `Square`, and `Rectangle` all extend `Shape`. `Shape` has a method `draw()`. All subclasses override `draw()`. A function to draw all elements in a list of `Shape` compatible objects

```
public static void drawAll(LinkedList<? extends Shape> l) {
        Object o;
        Iterator i = l.get_iterator();
        while (i.has_next()) {
                o = i.get_next();
                o.draw();
        }
}
```

- **Reflection**
  - Reflective programming: ability of a process to examine, introspect, and modify its own structure and behaviour
    - Introspection: can observe, and therefore reason about its own state.
      - example: can examine types

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

    - Intercession: can modify its execution state or alter its own interpretation or meaning.
  - To check if two variables of the same type in a generic function:

```
import java.lang.reflect.*;

class MyReflectionClass {
        public static boolean classequal(Object o1, Object o2) {
                return (o1.getClass() == o2.getClass());
        }
}
```

  - `getClass()` returns an object of type `Class` that encodes class information
  - Can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
...
```

  - 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();
...
```

    - more compactly: `Object o = Class.forName("Manager").newInstance()`
- Additional classes `Constructor`, `Method`, `Field`
    - Can use `getConstructors()`, `getMethods()` and `getFields()` to obtain constructors, methods and fields of `C` in an array
- Example: get 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();
        ...
}
```

- Can invoked methods and examine/set values of fields

```
...
Class c = obj.getClass();
...
Method[] methods = c.getMethods();
Object[] args = {...}; // Construct an array of arguments
methods[3].invoke(obj,args); // invoked methods[3] on obj with arguments arg

...

Field[] fields = c.getFields();
Object o = fields[2].get(obj); // get the value of fields[2] from obj
...
fields[3].set(obj,value); // set the value of fields[3] in obj to value
...
```

- Separate function to also include private components
    - `getDeclaredConstructors()`
    - `getDeclaredMethods()`
    - `getDeclaredFields()`
- `BlueJ`: a programming environment to learn Java
    - can define and compile Java classes
    - For compiled code, create object, invoke methods, examine state
- Limitation of Java reflection
    - cannot create or modify classes at run time
        - following is not possible: `Class c = new Class(...);`
    - An environment like `BlueJ` must invoked Java compiler before you can use a new class

Java Generics at Run-time

- Type erasure — Java does not keep record all of versions of `LinkedList<T>` as separate types
    - Cannot write: `if (s instanceof LinkedList<String>){...}`
- At run time, all type variables are promoted to `Object`
    - `LinkedList<T>` becomes `LinkedList<Object>`
    - Or, the upper bound, if one is available
        - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass()) { // condition is true
        ...
}
```

- If `S extends T` then `S[] extends T[]` but not the case in generics

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr; // Allowed. ETicker[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket(); // NOT Allowed. ticketarr[5] refers to an ETicket
```

  - To avoid, can declare a generic array, but cannot instantiate it

```
T[] newarray; // Allowed
newarray = new T[100]; // NOT Allowed
newarray = (T[]) new Object[100]; // Workaround
```

- Basic types `int, float, double, ...` are not compatible with `Object`. Wrapper class for each basic type

Basic type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long

Basic type	Wrapper Class
float	Float
double	Double
boolean	Boolean
char	Character

  - All wrapper classes other than `Boolean, Character` extend the class `Number`
- Wrapper classes
  - converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

  - Auto-boxing: implicit conversion between base types and wrapper types

```
int x = 5;
Integer myx = x;
int y = myx;
```

Week 6

- **Benefits of Indirection**
  - Two ways of implementing queues: Circular array and Linked list
    - Efficiency: Circular array is better – one time storage allocation
    - Flexibility: Linked list is better – circular array has bounded size
    - Offer user a choice of 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(){...};
        ...
}

CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq = new CircularArrayQueue<Date>();
stringq = new LinkedListQueue<String>();
```

- **What if we later realise we need a flexible size `dateq`?**
  - Implement an interface `Queue`

```
public interface Queue<E> {
        abstract void add (E element);
        abstract E remove();
        abstract int size();
}

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

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

Queue<Date> dateq;
Queue<String> stringq;

// Choice of implementation delayed to instantiation
dateq = new CircularArrayQueue<Date>();
stringq = new LinkedListQueue<String>();
```

Collections

- Java originally had pre-defined classes: `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
  - changing a choice requires multiple updates. Instead, organise these data structures by functionality
- **Collection interface**
  - `Collection` interface abstracts properties of grouped data: Arrays, lists, sets, etc. but not key-value structures
  - `add()` — add to the collection
  - `iterator()` — get an object that implements `Iterator` interface

```
public interface Collection<E>{
        boolean add(E element);
        Iterator<E> iterator();
        ...
}

public interface Iterator<E>{
        E next();
        boolean hasNext();
        void remove();
        ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
        String element = iter.next();
        // do something with the element
}
```

- Java later added "for each" loop — implicitly creates and iterator and runs through it

```
Collection<String> cstr = new ...;
for (String element : cstr)  {
        // do something with element
}

public static <E> boolean contains(Collection<E> c, Object obj) {
        for (E element : c) {
                if (element.equals(obj)) {
                        return true;
                }
        }
        return false
}
```

- `Collections` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`
  - To implement `Collection`, you need to implement all these methods!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from)
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

- `AbstractCollection` abstract class implements `Collection`
  - provides us with several of the functions and leaves us to focus only on the ones which we really normally would implement such as `add()`, `iterator()`, etc.

Concrete Collections

- Different data-types like Set, List, Queue are captured by interfaces that extend `Collection`
  - Interface `List` for ordered collections
  - Interface `Set` for collections without duplicates
  - Interface `Queue` for ordered collections with constraints on addition and deletion

The `List` interface

- Ordered collection can be accessed in two ways: through an iterator or by position
- Additional functions for random access
- `ListIterator` extends `Iterator`
  - `void add (E element)` to insert an element before the current index
  - `void previous()` to go to the previous element
  - `void hasPrevious()` checks that it is level to go backwards

```
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);
        ListIterator<E> listIterator();
}
```

- Random access is not efficient for all ordered collections: efficient for an array, but inefficient for a linked list
  - Tagging interface RandomAccess: tells us whether a `List` supports random access or not

```
if (c instanceof RandomAccess) {
        // use random access algorithm
} else {
        // use sequential access algorithm
}
```

- `AbstractCollection` is a "usable" version of `Collection`. Correspondingly, `AbstractList` extends `AbstractCollection`. `AbstractSequentialList` extends `AbstractList`.
- Concrete class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access; but random access methods of `AbstractList` are still available

```
LinkedList<String> list = new ...;

for (int i=0; i < list.size(); i++) {
        // do something with list.get(i);
}
```

- Concrete class `ArrayList<E>` extends `AbstractList`
- Two version of `add()`
  - `add()` from `Collection` appends to the end of the list
  - `add()` from `ListIterator` inserts a value before the current position of the iterator
- In `Collection`, `add()` returns `boolean`
  - `add()` may not update a set, always works for lists
- `add()` in `ListIterator` returns `void` because it always works

- **The `Set` interface**
  - A collection without duplicates
  - Identical to `Collection` but more constrained
    - `add()` should have no effect, and return `false` if the element already exists
    - `equals()` should return `true` if contents match after disregarding order
  - Map the value to its position: Hash function
  - Or arrange values in a two dimensional structure: Balanced search tree
  - `AbstractSet` extends `AbstractCollection`
  - Concrete sets
    - `HashSet` implements a hash table
      - underlying storage is an array
      - Map value `v` to a position `h(v)`

        - If `h(v)` is unoccupied, store `v` at that position. Otherwise, collision — different strategies to handle this case.
        - Unordered, but supports `iterator()`: scan elements in unspecified order, visit each element exactly once
    - `TreeSet` uses a tree representation
        - Values are ordered
        - Sorted collection
        - Iterator will visit elements in sorted order
        - Insertion is more complex than a hash table: Time $O(\log n)$ if the set has n elements.
- **The `Queue` interface**
    - Ordered, remove front, insert rear
    - `Queue` interface supports: `boolean add(E element)` and `E remove()`
        - If queue full, `add()` flags an error
        - If queue empty, `remove()` flags an error
    - Gentler `add()` and `remove()`: `boolean offer(E element)` and `E poll()`
        - Return `false` or `null` if not possible
    - Inspect the head, no update: `E element()` and `E peak()`
- **The `Deque` interface**
    - functionalities of a double ended queue:

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

- **The `PriorityQueue` interface**
    - `remove()` returns highest priority item

- **Maps**
    - `Map` interface
        - Two type parameters

```
public interface Map<K,V> {
        V get (Object key);
        V put (K key, V Value);

        boolean containsKey(Object key);
        boolean containsValue(Object value);
        ...
}
```

        - keys form a set i.e., no duplicates
            - `put(k,v)` returns the previous value associated with `k`, or `null`
    - Updating a map

- Key-value stores are useful to accumulate quantities: frequency of words, total runs, etc.
- Initialisation problem: update the value if the key exists, otherwise create a new entry
- `Map` has the following default method: `V getOrDefault(Object key, V defaultValue)` : if there is a value in for that key then return that value, if the key doesn't exist, return the default value
- `putIfAbsent()` to initialise a missing key; alternative to `getOrDefault()`
- `merge` initialise to `newscore` if no key `bat` otherwise combine current value with `newscore` using `Integer::sum` (sum function from the Integer class)

```
Map<String, Integer> scores = ...;
int score = scores.getOrDefault(bat, 0);
scores.put(bat, scores.getOrDefault(bat,0)+newscore);

scores.putIfAbsent(bat,0);
scores.put(bat,score.get(bat)+newscore);

scores.merge(bat, newscore, Integer::sum);
```

- Extracting keys and values
  - `Set<K> keySet()` returns the keys
  - `Collection<V> values()` returns the values
  - `Set<Map.Entry<K,V>> entrySet()` returns the key-value pairs
  - Key-value pairs from a set over a special type `Map.Entry`
  - Can iterate through a Map

```
Set<String> keys = strmap.keySet();
for (String key : keys) {
        // do something with key
}
```

```
// Iteration through key-value pairs
for (Map.Entry<String,Employee> entry : staff.entrySet()) {
        String k = entry.getKey();
        Employee v = entry.getValue();
        // do something with k, v
}
```

- Concrete implementation of Map
  - **`HashMap`**
    - similar to `HashSet`
    - Use a hash table to store keys and values
    - No fixed order over keys returned by `keySet()`
  - **`TreeMap`**
    - Similar to `TreeSet`
    - balanced search tree to store keys and values
    - Iterator over `keySet()` will process keys in sorted order
  - **`LinkedHashMap`**
    - Remembers the order in which keys were inserted
    - Hash table entries are also connected as a (doubly) linked list
    - Iterators over both `keySet()` and `value()`
    - can also use access order instead of insertion order

- Each `get()` or `put()` moved key-value pair to end of list

Week 7

- **Dealing With Errors**
  - Exception Handling
    - Code that generates error `raises` or `throws` an `exception`
    - Notify the type of error: nature of the exception; structure an exception as an object
    - Caller catches the exception and takes corrective action: extract information; graceful interruption rather than program crash
  - Java's classification of errors
    - All exception descend from class `Throwable`
      - Two branches: `Error` and `Exception`
    - `Error` — relatively rare, "not the programmer's fault"
      - internal error, resource limitations within runtime
      - no realistic corrective action possible
    - `Exception` — two branches
      - `RunTimeException`: programming errors (array index out of bounds, invalid hash key, etc.)
      - Checked exception: typically user-defined, code assumptions violated
- **Exceptions in Java**
  - `try-catch`
    - enclose code that may generate exception in `try` block

```
try {
        // call a function that may throw an exception
}
catch (ExceptionType e) {
        // examine e and handle it
}
```

    - can catch more than one type of exception: multiple `catch` blocks
    - Exceptions are classes in Java class hierarchy: `catch (ExceptionType e)` matches any subtype of `ExceptionType`
    - Catch blocks are tried in sequence; order `catch` blocks by argument type, more specific to less specific
    - Notifying checked exceptions
      - Example: you write a method `readData()`, which expected 2048 lines but actual data is less than promised length
      - Throw `EOFException`, subtype of `IOException`: signals that EOF has been reached unexpectedly during input
      - `throw new EOFException()`
      - can also pass message when constructing exception object

```
String errormsg = "Content-Length:" + contentlen + ", Received: " + rcvdlen;
throw new EOFException(errormsg);
```

    - Throwing exceptions
      - How does called know that `readData()` generates `EOFException`
      - Declare exceptions thrown in header

```
String readData(Scanner in) throws EOFException, FileNotFoundException {
        ...
        while(...) {
                if (!in.hasNext()) {
```

```
                        // EOF encountered
                        if (n < len) {
                                String errmsg = ...
                                throw new EOFException(errmsg);
                        }
                }
        }
}
```

  - Can throw any subtype of declared exception type

- **Customised Exceptions**
  - example: don't want negative numbers in a `LinearList`; define a new class extending `Exception`

```
public class NegativeException extends Exception {
        private int error_value;
        public NegativeException(String message, int i) {
                super(message);
                error_value = i;
        }
        public int report_error_value() {
                return error_value;
        }
}

public class LinearList {
        ...
        public add(int i) throws NegativeException {
                ...
                if (i<0) {
                        throw new NegativeException("Negative input", i);
                }
        }
}
```

  - to extract information about the exception where the function was called:

```
try {
        ...
        // call a fuction that may throw NegativeException
}
catch (ExceptionType e) {
        ...
        String errormsg = e.getMessage();
}
```

  - chaining exceptions: process and throw a new exception from `catch`

```
try {
        ...
        // access database
}
catch (SQLException e) {
        ...
        String errormsg = "database error" + e.getMessage();
        throw new ServletException(errormsg);
}
```

  - `Throwable` has additional methods to track chain of exceptions: `getCause()`, `initCause()`

```
try {
        ...
        access database
}
catch (SQLException e) {
        ...
        String errormsg = "database error" + e.getMessage();
        ServletException new_e = new ServletException(errormsg);
        new_e.initCause(e);
        throw new_e;
}
```

```
try {
        ...
}
catch (ServletException e) {
        ...
        Throwable original = e.getCause();
}
```

- Cleaning up resources: `finally { ... }` block

- **Packages**
  - organisational unit called `package`
  - can use `import` to use packages directly
    - `import java.match.BigDecimal`
    - for all classes in `.../java/math` : `import java.math.*`
  - `*` is not recursive; cannot write `import java.*`
  - can create our own hierarchy of packages
  - Add a package header to include a class in a package

```
package in.ac.iitm.onlinedegree;
public class Employee { ... }
```

  - By default, all classes in a directory belong to same anonymous package
  - If we don't use modifiers `public` or `private`, the default visibility is public within the package
    - for both methods and variables
  - Can restrict visibility with respect to inheritance hierarchy
    - `protected` means visible within the subtree, so all subclasses
    - normally, a subclass cannot expand the visibility but if the parent class is `protected`, the child class can make itself `public`

- **Assertions**
  - Function may have constraints on the parameters

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

  - If assertion fails, code throws `AssertError`
  - can provide additional information to be printed with diagnostic message `:`
  - Assertions are enable or disables at runtime: does not require recompilation
  - use following flag to run with assertions enabled: `java -enableassertions MyCode` ; can use `-ea` as abbreviation
  - can selectively turn on assertions for a class: `java -ea:MyClass MyCode`
    - or a package: `java -ea:in.ac.iitm.onlinedegree MyCode`

- disable assertions: `java -disableassertions MyCode`
- can combine: `java -ea:in.ac.iitm.onlinedegree -da:MyClass MyCode` (enable for the package but disable for a particular class)
- separate switch to enable assertions for system class: `java -enablesystemassertions MyCode` ; or `java -esa MyCode`

- **Logging**
  - Typical to generate messages within code for diagnosis
  - Log diagnostics messages separately
    - Logs are arranged hierarchically — choose the level of logging needed
    - Logs can be processed by other code — handlers
    - Logging can be controlled by a configuration file
  - Simplest: call `info()` method of global logger: `Logger.getGlobal().info("Edit->Copy menu item selected");`
  - Suppress logging by executing: `Logger.getGlobal().setLevel(Level.OFF);`
  - Create a custom logger:

```
private static final Logger myLogger = Logger.getLogger("in.ac.iitm.onlinedegree");
```

    - Logger names are hierarchical, like package names
    - Setting a property for `in.ac.iitm` automatically sets it for `in.ac.iitm.onlinedegree`
  - Seven logging levels:
    - `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: `logger.setLevel(Level.ALL);`
    - Turn off all logging: `logger.setLevel(Level.OFF);`
  - Can change logging properties through a configuration file

Week 8

- **Cloning**
  - **Soft copy — Bitwise copy**
    - `Object` defines a method `clone()` which returns a bitwise copy of an instance.

```
public class Employee {
        private String name;
        private double salary;
        private Date bday;
        ...
        public void setName(String n) {
                this.name = n;
        }

        public void setBday(int dd, int mm, int yy) {
                bday.update(dd,mm,yy);
        }
}
```

```
Employee e1 = new Employee("Dhruv", 25100);
Employee e2 = e1.clone();
```

```
e2.setName("Akul");     // e1 will not be impacted
e2.setBday(16,4,1997);  // e1 also changed!
```

- problem with bitwise copy is that it cannot make sure that the embedded objects are also copied bitwise!
- Bitwise copy is a shallow copy

- **Deep copy**
  - recursively cloned nested objects
  - Override the shallow `clone()` from `Object`

```
public class Employee {
        ...
        public Employee clone() {
                Employee newemp = (Employee) super.clone();
                Date newbday = (Date) bday.clone();
                newemp.bday = newbday;
                return newemp;
        }
        ...
}
```

  - If `Manager` extends `Employee`

```
public class Manager extends Employee {
        private Date promoDate;
}
```

    - `Manager` inherits deep copy `clone()` from `Employee`. However `Employee.clone()` does not know that it has to deep copy `promoDate`

- To allow `clone()` to be used, a class has to implement `Cloneable` interface

```
public class Employee implements Cloneable {
        private String name;
        private double salary;
        private Date bday;
        ...
        public Employee clone() throws CloneNotSupportedException {
                ...
        }
}
```

```
Employee e1 = new Employee("Dhruv", 25000);
Employee e2 = e1.clone();
```

  - `Cloneable` is a Market interface; it doesn't have any implementable functions, just a boolean value
  - `clone()` in `Object` is protected i.e., only `Employee` objects can call `clone()`
  - Redefine `clone()` as `public` to allow other classes to clone `Employee`
  - `CloneNotSupportedException` is thrown when the class doesn't implement `Cloneable`
  - Always use `clone()` in a try block

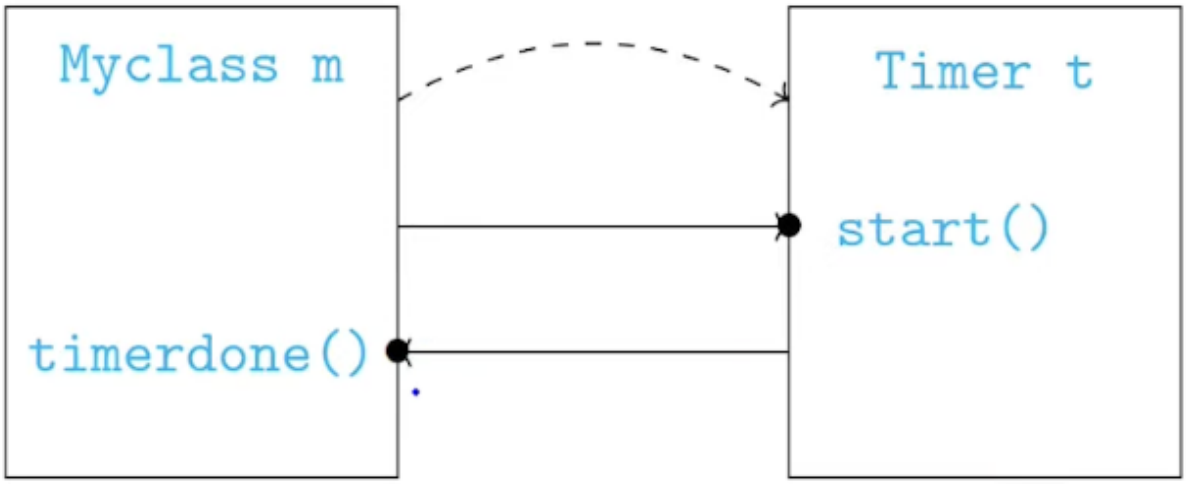
Type Inference

- Java insists that all variables are declared in advance. The compiler can then check wether the program is well-typed.
- An alternative approach is to do "type inference"
  - Assume code is well-typed, derive most general types: use information from constants to determine type
- Depending on what functions are being invoked, you can distinguish between an object and a subclass

- Java allows limited type inference
  - only for local variables in functions
  - not for instance variables of a class
  - use generic `var` to declare variables; must be initialised when declared, type is inferred from initial value

```
var b = false;  // boolean
var s = "Hello, World";  // String
var d = 2.0;  // double
var f = 3.14f;  // float
var e = new Manager(...);  // Manager
```

- **Higher Order Functions**
  - Higher order function: a function that takes another function as an argument
  - `Myclass m` creates a `Timer t` and `t` starts running in parallel; `t` notifies `m` when the time limit expires.



  - `m` needs to pass `timerdone()` to `t`
  - Achieved through interface

```
public interface Timerowner {
        public abstract void timerdone();
}

public class Myclass extends Timerownder{
        ...
}

public class TImer implements Runnable {
        private Timerowner owner;
        ...
        public void start() {
                ...
                owner.timerdone();
        }
}
```

    - Instead of passing a function directly, pass the function through an object which knows how to implement the function

- Customise `Arrays.sort`
  - `Comparator` interface provides for comparison function

```
public interface Comparator<T> {
        public abstract int compare( T o1, T o2);
}
```

- Implement `Comparator`

```
public class StringCompare implements Comparator<String> {
        public int compare (String s1, String s2) {
                return s1.length() - s2.length();
        }
}
String[] strarr = new ...;
Arrays.sort(strarr,StringCompare);
```

- Functional interfaces: interfaces that define a single function: `TimerOwner` or `Comparator`
- **Lambda expressions**
  - anonymous functions; return value and type are implicit
  - (Parameters) → Body
  - `(String s1, String s2) -> s1.length() - s2.length()`

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

  - More complicated function body can be defined as a block

```
(String s1, String s2) -> {
        if s1.length() < s2.length()
                return -1;
        else if s1.length() > s2.length()
                return 1;
        else
                return 0;
        }
```

  - The function that received the lambda expression needs to use a functional interface for the parameter type

```
public static <T> void Array.sort(T[] a, Comparator<T> c)
```

- Passing named functions
  - If lambda expression consists of a single function call, we can pass that function by name

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

  - Expression should call a function, and nothing else
- Method references
  - `ClassName::StaticMethod`
    - Method reference is `C::f`
    - Corresponding expression with as many arguments as `f` has
  - `ClassName::InstanceMethod`
    - Called with respect to an object that becomes implicit parameter
  - `object::InstanceMethod`
    - Arguments are passed to `o.f`
  - can also pass references to constructors

Streams

- Operating on collections
  - Usually use an iterator to process a collection

- Alternative approach — generate a stream of values from a collection

```
List<String> words = ...;
long count = 0;
for (String w : words) {
        if (w.length() > 10) {
                count++;
        }
}
```

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

// Processing can be parallelized
long count = words.parallelStream()
                        .filter(w -> w.length() > 10)
                        .count();
```

  - Stream processing is declarative; focus on what to compute, rather than how
- Lazy evaluation is possible
  - If we want first 10 long words; stop generating the stream once we find 10 such words
  - need not generate the entire steam in advance
- Working with streams
  - create a stream
  - pass through intermediate operations that transform streams
  - apply a terminal operation to get a result
- Stream operations are non-destructive: input stream is untouched
- `stream()` is a part of `Collections` interface
- Static method `Stream.generate()` generates a stream from a function; provide a function that produces values on demand
- `Stream.iterate()` — stream of dependent values
  - initial value, function to generate the next value from the previous one

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos = Stream.generate(() -> "Echo");

Stream<Double> randomds = Stream.generate(Math::random);

Stream<Integer> integers = Stream.iterate(0, n-> n+1);

Stream<Integer> integers = Stream.iterate(0, n-> n < 100, n -> n+1);
```

- `map()` applies a function to each element in the stream
  - Extract the first letter of each long word

```
Stream<String> startlongwords = words.stream()
                        .filter(w -> w.length() > 10)
                        .map(s -> s.substring(0,1));
```

- `flatMap()` flattens (collapses) nested list into a single stream

```
Stream<String> longwords = words.stream()
                        .filter(w -> w.length() > 10)
                        .flatMap(s -> explode(s));
```

- Make a stream finite — `limit(n)` : Generate 100 random numbers

```
Stream<Double> randomds = Stream.generate(Math::random).limit(100);
```

- Skip `n` elements — `skip(n)` : Discard the first 10 number:

```
Stream<Double> randomds = Stream.generate(Math::random).limit(10);
```

- Stop when element matches a criterion — `takeWhile()` : Stop with number smaller than 0.5

```
Stream<Double> randomds = Stream.generate(Math::random).takeWhile(n -> n >= 0.5);
```

- Start after element matches a criterion — `dropWhile()` : Start after getting a number larger than 0.05

```
Stream<Double> randomds = Stream.generate(Math::random).dropWhile(n -> n <= 0.05);
```

- Number of elements — `count()`
- Largest and smallest values seen — `max()` and `min()` ; requires a comparison function

```
Optional<Double> maxrand = Stream.generate(Math::random)
                                                    .limit(10)
                                                    .max(Double::compareTo);
```

- First element — `findFirst()`

```
Optional<Double> firstrand = Stream.generate(Math::random)
                                                    .limit(100)
                                                    .filter(n -> n > 0.999)
                                                    .findFirst();
```

Week 9

- **Optional Types**
  - `max()` of an empty stream in undefined. Return value could be `Double` or `null`
  - `Optional<T>` object
    - Wrapper
    - May contain an object of type `T` is value if present; or no object
  - Handling missing optional values
    - `orElse()` to pass a default value
    - `orElseGet()` to call a function to generate replacement
    - `orElseThrow()` to generate an exception when a missing value is encountered

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

```
Double fixrand = maxrand.orElse(-1.0);
Double fixrand2 = maxrand.orElseGet(() -> SomeFunctionToGenerateDouble());
Double fixrand3 = maxrand.orElseThrow(IllegalStateException::new);
```

- Ignoring missing values
  - `ifPresent()` to test if a value is present, and process it; missing value is ignored
  - `ifPresentOrElse()` specify an alternative action if the value is not present

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

var results = new ArrayList<Double>();
maxrand.ifPresent(v -> results.add(v)); // lambda function
maxrand.ifPresent(results:add); // lambda function

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

- Creating an optional value
  - `Optional.of(v)` creates a value `v`
  - `Optional.empty()` creates empty optional

```
public static Optional<Double> inverse(Double x) {
        if (x==0) {
                return Optional.empty();
        } else {
                return Optional.of(1/x);
        }
}
```

  - Use `ofNullable()` to transform `null` automatically into an empty optional

```
public static Optional<Double> inverse(Double x){
        return Optional.ofNullable(1/x);
}
```

- Passing on optional values
  - `map` applies function to value, if present; if input is empty, so it output
  - `or` if value is present it is passed as is; if not, value generated by `or()` is passed

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

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

var results = new ArrayList<Double>();
maxrand.map(results::add);

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

- Example: some function `f()` returns `Optional<T>`. Class `T` defines `g()`, returning `Optional<U>`
  - Cannot compose `s.f().g()` because `s.f()` has type `Optional<T>` not `T`
  - Use `flatMap()`
    - If `s.f()` is present, apply `g()` otherwise return empty `Optional<U>`

```
Optional<U> result = s.f().flatMap(T::g);
```

- pass output of safe `inverse` to `squareRoot()`

```
public static Optional<Double> inverse(Double x) {
        if (x==0) {
                return Optional.empty();
        } else {
                return Optional.of(1/x);
        }
}

public static Optional<Double> squareRoot(Double x) {
        if (x<0) {
                return Optional.empty();
        } else {
                return Optional.of(Math.sqrt(x));
        }
}

Optional<Double> result = inverse(x).flatMap.(MyClass::squareRoot);
```

- Turning an optional into a stream
  - `lookup(u)` returns a `User` if `u` is a valid username
  - want to convert a stream of userids into a stream of users
    - input: `Stream<String>`
    - output: `Stream<User>`
  - pass through a `flatMap()`

```
public Optional<User> lookup(String id) {...}

Stream<String> ids = ...;
Stream<User> users = ids.map(Users::lookup).flatMap(Optional::stream);
```

  - If `oldlookup()` was implemented without using `Optional` i.e., it either returns a null or a `User`

```
public User oldlookup(String id) {...}

Stream<String> ids = ...;
Stream<User> users = ids.flatMap( id -> Stream.ofNullable(Users.oldlookup(id)) );
```

Collecting Results From Streams

- collecting results

```
mystream.forEach(System.out::println); // to iterate on the Stream

Object[] result = mystream.toArray(); // convert the stream into an Array
String[] result = mystream.toArray(String[]::new);
```

- Storing a stream as a collection
  - Stream is created from a collection. to go back from stream to collection
  - use `collect()`
    - pass appropriate factory method from `Collectors`

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

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

- Stream summaries
  - `count()`, `max()`, `min()`
  - `Collectors` has methods to aggregate summaries in a single object
    - `summarizingInt` works for a stream of integers. Pass function `String::length` to convert the stream of `String` to numbers
    - Returns `IntSummaryStatistics` that stores count, max, min, sum, average
    - `getCount()`, `getMax()`, `getMin()`, `getSum()`, `getAverage()`

```
IntSummaryStatistics summary = mystream.collect(

Collectors.summarizingInt(String::length)
                                                                    );

double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

  - similarly `summarizingLong()` and `summarizingDouble()` return `LongSummaryStatistics` and `DoubleSummaryStatistics()`
- Converting a stream to a map
  - example: convert a stream of `Person` to a map. For `Person p`, `p.getID()` is key and `p.getName()` is value
  - to store entire object as value: `Function.identity()`
  - when the index is is not unique, provide a function to fix, for e.g. in the case of `nameToID`

```
Stream<Person> people = ...;

Map<Integer, String> idToName = people.collect(Collectors.toMap(

Person::getId,

Person::getName

));

Map<Integer, Person> idToPerson = people.collect(Collectors.toMap(

Person::getId,

Function.identity()

));

Map<String, Integer> nameToID = people.collect(Collectors.toMap(

Person::getName,

Person::getId,

(existingValue, newValue) ->

existingValue

));
```

  - Instead of discarding values with duplicate keys, group them

```
Map<String, List<Person>> nameTopersons = people.collect(

Collectors.groupingBy(

Person::getName

));
```

- may want to partition the stream using a predicate. Partitioning names into those that start with "A" and the rest

```
Map<Boolean, List<Person>> aAndOtherPersons = people.collect(

Collectors.partitioningBy(
                                                                    p
-> p.getName().substr(0,1).equals("A")
                                                                ));

List<Person> startingLetterA = aAndOtherPersons.get(true);
```

- **Input/Output Streams**
  - Input: read a sequence of bytes from some source (a file, internet connection, memory)
  - Output: write a sequence of bytes to some source (a file, internet connection, memory)
  - these input/output are referred to as streams (Not the same as `Stream` class)
  - Ultimately, input and output are raw uninterpreted bytes of data
  - Use a pipeline of input/output stream transformers
  - Reading and writing raw bytes
    - Classes `InputStream` and `OutputStream`
    - Read one or more bytes
    - Example functions available in `InputStream`

```
abstract int read();
int read(byte[] b);
byte[] readAllBytes();
// ... and more

// check availability before reading
InputStream in = ...
int bytesAvailable = in.available();
if (bytesAvailable > 0) {
        var data = new byte[bytesAvailable];
        in.read(data);
}
```

    - Example functions available in `OutputStream`
      - Close a stream when done — release resources
      - Flush an output stream — output is buffered

```
abstract void write(int b);
void write(byte[] b);
// ... and more

OutputStream out = ...
byte[] values = ...;
out.write(values);

in.close();
```

```
out.flush();
```

- Connecting a stream to an external source
  - Create an input stream attached to a file
  - Create an output stream attached to a file

```
var in = new FileInputStream("input.class");
var out = new FileOutputStream("output.bin");

var out = new FileOutputStream("output.bin", false); // Overwrite
var out = new FileOutputStream("output.bin", true); // Append
```

- Reading and writing text
  - `Scanner` class

```
var fin = new FileInputStream("input.txt");
var scin = new Scanner(fin);

var scin = new Scanner( new FileInputStream("input.txt") );

String s = scin.nextLine(); // One Line
String w = scin.next(); // One word
int i = scin.nextInt(); // Read an int
boolean b = scin.hasNext(); // Any more words?
```

  - `PrintWriter` class for writing text

```
var fout = new FileOutputStream("output.txt");
var pout = new PrintWriter(fout);

var pout = new PrintWriter( new FileOutputStream("output.txt") );

String msg = "Hello, world!";
pout.println(msg);
```

  - Copy input text file to output text file

```
var in = new Scanner(...);
var out = new PrintWriter(...);

while (in.hasNext()) {
        String line = in.nextLine();
        out.println(line);
}
```

- Reading and writing binary data
  - `DataInputStream` class; can apply to any input stream; many read methods

```
var fin = new FileInputStream("input.txt");
var din = new DataInputStream(fin);

var din = new DataInputStream( new FileInputStream("input.txt") );

readInt, readShort, readLong,
readFloat, readDouble,
readChar, readUTF,
readBoolean
```

- `DataOutputStraem` class

```
var fout = new FileOutputStream("output.txt");
var dout = new DataOutputStraem(fout);

var dout = new DataOutputStraem( new FileOutputStream("output.txt") );

writeInt, writeShort, writeLong,
writeFloat, writeDouble, writeChar,
writeUTF, writeBoolean, writeChars, writeByte
```

- copy input binary file to output binary file

```
var in = new DataInputStream(...);
var out = new DataOutputStream(...);

int bytesAvailable = in.available();
while (bytesAvailable > 0) {
        var data = new byte[bytesAvailable];
        in.read(data);
        out.write(data);
        bytesAvailable = in.available();
}
```

- Buffering an input stream
  - reads blocks of data, more efficient

```
var din = new DataInputStream(
                                new BufferedInputStream(
                                        new FileInputStream("grades.dat")
                                        )
                                );
```

- Speculative reads
  - examine the first element, return to stream if necessary

```
var pbin = new PushbackInputStream(
                                new BufferedInputStream(
                                        new FileInputStream("grades.dat")
                                        )
                                );
int b = pbin.read();
if (b != '<') {pbin.unread(b)};
```

- to use `PushbackInputStream` to take input data

```
var pbin = new PushbackInputStream(
                                new BufferedInputStream(
                                        new FileInputStream("grades.dat")
                                        )
                                );

var     din = new DataInputStream(pbin);
```

- **Serialisation**
  - Serialization and deserialization: writing and reading objects
  - `ObjectOutputStream` to write objects
    - examines all the fields and saves their contents

```
var out = new ObjectOutputStream( new FileOutputStream("employee.dat") );

var emp = new Employee(...);
var boss = new Manager(...);
out.writeObject(emp);
out.writeObject(boss);
```

- `ObjectInputStream` to read objects
  - "reconstructs" the object, effectively calls a constructor

```
var out = new ObjectInputStream( new FileInputStream("employee.dat") );

var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

- Class has to allow serialization — implement market interface

```
public class Employee implements Serializable {...}
```

- In the case of a nested object:

```
class Manager extends Employee {
        private Employee secretary;
        ...
}
```

  - Two managers might have the same secretary. how to avoid duplicating objects?
  - implicitly, each object is assigned a serial number. If saved previously, record serial number instead of saving saving the object again
- Mark fields as `transient` if you don't want the object to be serialized i.e., you want to save duplicate copies
  - need to override `writeObject()` and `readObject()`

```
public class LabeledPoint implements Serializable {
        private String label;
        private transient Point2D.Double point;

        private void writeObject (ObjectOutputStream out) throws IOException{
                out.defaultWriteObject();
                out.writeDouble(point.getX());
                out.writeDouble(point.getY());
        }

        private void readObject(ObjectInputStream in) throws IOException {
                in.deaultReadObject();
                double x = in.readDouble();
                double y = in.readDouble();
                point = new Point2D.Double(x,y);
        }
}
```

- Older serialized objects may be incompatible with newer versions
- Deserialization implicitly invokes a constructor; running a code from an external source is a security risk

Week 10

- **Concurrency**
  - Multiprocessing: time-slicing to share access
  - Logical parallel actions within a single application
    - Clicking `Stop` terminates a download in a browser; User-interface is running in parallel with network access
  - Process
    - Private set of local variables
    - Saving the state of one process and loading the suspended state of another
  - Threads
    - Operated on same local variables
    - Communicate via 'shared memory'
    - Context switches are easier
  - Shared variables
    - browser example: download thread and user-interface thread run in parallel
      - shared boolean variable `terminate` indicates whether download should be interrupted
    - Watch out for race conditions: when many threads are trying to change the same variable
      - shared variables must be updated consistently
  - Creating threads: Have a class extend `Thread`

```
public class Parallel extends Thread{
        private int id;
        public Parallel(int i) { id = i;}
        public void run(){
                for (int j=0; j<100, j++) {
                        System.out.prinln("my id: " + id);
                        try{
                                sleep(1000);
                        }
                        catch(InterruptedException e){}
                }
        }
}

public class Test {
        public static void main (String[] args) {
                Parallel p[] = new Parallel[5];
                for (int i=0; i<5; i++) {
                        p[i] = new Parallel(i);
                        p[i].start(); // Start p[i].run() in concurrent thread
                }
        }
}
```

  - Cannot always extend `Thread` because Java only allows one class extension
    - Instead, implement `Runnable`
    - Have the create `Thread` objects

```
public class Parallel implements Runnable{
        // only the line above has changed
        private int id;
        public Parallel(int i) {...} // Constructor
        public void run() {...}
}

public class Test{
        public static void main (String[] args) {
                Parallel p[] = new Parallel[5];
                Thread t[] = new Thread[5];

                for (int i=0; i<5; i++) {
```

```
                        p[i] = new Parallel[i];
                        t[i] = new Thread(p[i]); // Make a thread t[i] from p[i]
                        t[i].start();
                }
        }
}
```

- **Race Conditions**
  - Maintaining data consistency
    - `double accounts[100]` describes 100 bank accounts
    - two functions operate on `accounts`: `transfer()` and `audit()`

```
boolean transfer (double amount, int source, int target) {
        if (accounts[source] < amount) {
                return false;
        }
        accounts[source] -= amount;
        accounts[target] += amount;
        return true;
}

double audit() {
        double balance = 0.0;
        for (int i=0; i<100; i++){
                balance += accounts[i];
        }
        return balance;
}
```

    - if a transfer is made in a thread and audit is called in another thread, the audit figure might not be the true value because the first thread is still processing
  - 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
- **Mutual Exclusion**
  - mutual exclusion for two processes

```
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;
...
```

    - Starvation: one thread is locked out permanently if other thread shuts down

- mutual exclusion using two boolean variables

```
Thread 1                          Thread 2
...                               ...
request_1 = true;                 request_2 = true;
while (request_2){                while (request_1)
  // "Busy" wait                    // "Busy" wait
}                                 }
// Enter critical section         // Enter critical section
   ...                               ...
// Leave critical section         // Leave critical section
request_1 = false;                request_2 = false;
...                               ...
```

  - Deadlock: if both threads try simultaneously, they block each other
- **Peterson's algorithm**
  - If both try simultaneously, `turn` decides which goes through
  - If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

```
Thread 1                          Thread 2
...                               ...
request_1 = true;                 request_2 = true;
turn = 2;                         turn = 1;
while (request_2 &&               while (request_1 &&
       turn != 1){                       turn != 2){
  // "Busy" wait                    // "Busy" wait
}                                 }
// Enter critical section         // Enter critical section
   ...                               ...
// Leave critical section         // Leave critical section
request_1 = false;                request_2 = false;
...                               ...
```

- Lamport's Bakery Algorithm — for n process mutual exclusion
  - each new process picks up a token (increments a counter) that is larger than all waiting processes
  - lowest number gets served next
  - still need to break ties — token counter is not atomic

Test and Set

- Fundamental issue preventing consistent concurrent updates of shared variables: test-and-set
- To increment a counter, check its current value, then add 1. If more than one thread does this is in parallel, updates may overlap and get lost
- **Dijkstra's Semaphores**
  - Integer variable with atomic test-and-set operation
  - A semaphore `S` supports two atomic operations
    - `P(s)` — to pass

```
if (S > 0) // test
        decrement S; // set
else
        wait for S to become positive // wait
```

    - `V(s)` — to release

```
if (there are threads waiting for S to become positive)
    wake one of them up;
            // choice is nondeterministic
else
        increment S;
```

- Mutual exclusion using semaphores

```
Thread 1                          Thread 2
...                               ...
P(S);                             P(S);
// Enter critical section         // Enter critical section
   ...                               ...
// Leave critical section         // Leave critical section
V(S);                             V(S);
...                               ...
```

  - guarantees mutual exclusion; freedom from starvation and deadlock

- **Monitors**
  - Attach synchronisation control to the data that is being protected
  - like a class in an OO language
    - data definition — to which access is restricted across threads
    - collections of functions operating on this data — all are implicitly mutually exclusive
    - guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish
  - Implicit queue associated with each monitor: contains all processes waiting for access
  - Have a separate internal queue, as opposed to external queue where initially blocked threads wait
    - dual operation to notify and wake up suspended processes

```
monitor bank_account {
        ...
        boolean transfer (double amount, int source, int target) {
                if (accounts[source] < amount) { wait(); }
                accounts[source] -= amount;
                accounts[target] += amount;
                notify();
                return true;
        }
}
```

    - Signal and exit — notifying process immediately exits the monitor — `notify()` must be the last instruction
    - Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor
    - Signal and continue — notifying process keeps control till it completes and then one of the notified processes steps in
  - Should check `wait()` condition again on wake up — change of state may not be sufficient to continue

```
monitor bank_account {
        ...
        boolean transfer (double amount, int source, int target) {
                while (accounts[source] < amount) { wait(); }
                accounts[source] -= amount;
                accounts[target] += amount;
                notify();
                return true;
        }
}
```

- Makes sense to have more than one internal queue. Pseudo code:

```
monitor bank_account {
        double accounts[100];
        queue q[100]; // one internal queue for each account

        boolean transfer(double amount, int source, int target) {
                while (accounts[source] < amount) {
                        q[source].wait(); // wait in the queue associated with source
                }
                accounts[source] -= amount;
                accounts[target] += amount;
                q[target].notify(); // notify the queue associated with target
                return true;
        }

        double audit() {
                // compute the balance across all accounts
        }
}
```

Week 11

- **Monitors**
  - Incorporated within existing class definitions; every class can behave like a monitor
  - Use `synchronized` modifier in the function definitions
  - Each object has a lock. To execute a `synchronized` method, thread must acquire lock. Thread gives up lock when the method exits. Only one thread can have the lock at any time.

```
public class bank_account{
        double accounts[100];

        public synchronized boolean transfer(double amt,
                                                                        int
source,
                                                                        int
target){
                while (accounts[source] < amt) { wait();}
                accounts[source] -= amt;
                accounts[target] += amt;
                notifyAll();
                return true;
        }

        public synchronized double audit() {
                double bbalance = 0.0;
                for (int i=0; i<100; i++){
                        balance += accounts[i];
                }
                return balance;
        }

        public double current_balance(int i) {
                return accounts[i];
        }
}
```

  - Object locks
    - `f()` and `g()` can start in parallel but only one of the threads can grab the lock for `o`

- each object has its own internal queue

```
public class XYZ{
        Object o = new Object();

        public int f() {
                ...
                synchronized(o){
                        ...
                        o.wait();
                        ...
                }
        }

        public double g() {
                ...
                synchronized(o){
                        ...
                        o.notifyAll();
                        ...
                }
        }

        public double h() {
                synchronized(this){
                ...
                }
        }
}
```

- `wait()` can be interrupted. Should write `wait` in `try-catch` block

```
try{
        wait();
} catch (InterruptedException e) {
        ...
};
```

- `IllegalMonitorStateException` : Error to use `wait()`, `notify()`, `notifyAll()` outside `synchronized` method
- `ReentrantLock` class
  - Similar to semaphore

```
public class Bank {
        private Lock bankLock = new ReentrantLock();
        ...
        public void transfer (int from, int to, int amt) {
                bankLock.lock();
                try {
                        accounts[from] -= amt;
                        accounts[to] += amt;
                }
                finally {
                        bankLock.unlock();
                }
        }
}
```

- **Threads**
  - Creating threads: Have a class extend `Thread`

```
public class Parallel extends Thread{
        private int id;
        public Parallel(int i) { id = i;}

        public void run(){
                for (int j=0; j<100, j++) {
                        System.out.prinln("my id: " + id);
                        try{
                                sleep(1000);
                        }
                        catch(InterruptedException e){}
                }
        }
}

public class Test {
        public static void main (String[] args) {
                Parallel p[] = new Parallel[5];
                for (int i=0; i<5; i++) {
                        p[i] = new Parallel(i);
                        p[i].start(); // Start p[i].run() in concurrent thread
                }
        }
}
```

- Creating threads: Have a class implement `Runnable`

```
public class Parallel implements Runnable{
        // only the line above has changed
        private int id;
        public Parallel(int i) {...} // Constructor
        public void run() {...}
}

public class Test{
        public static void main (String[] args) {
                Parallel p[] = new Parallel[5];
                Thread t[] = new Thread[5];

                for (int i=0; i<5; i++) {
                        p[i] = new Parallel[i];
                        t[i] = new Thread(p[i]); // Make a thread t[i] from p[i]
                        t[i].start();
                }
        }
}
```

- Life cycle of a Java thread — thread status via `t.getState()`
  1. New: Created but not `start()` ed
  2. Runnable: `start()` ed and ready to be scheduled
     - need to be actually running; no order of scheduling; use time-slicing
     - Not available to run
  3. Blocked: waiting for a lock, unblocked when lock is granted
  4. Waiting: suspended by `wait()`, unblocked by `notify()` or `notifyAll()`
  5. Timed wait: within `sleep(...)`, released when sleep time expires
  6. Dead: thread terminates
- Interrupts
  - One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
  - Raises `InterruptedException` within `wait()` or `sleep()`

- No exception if the thread is running — `interrupt()` set a status flag — `interrupted()` checks interrupt status and clears the flag

```
public void run() {
        try{
                j=0;
                while (!interrupted() && j < 100) {
                        System.out.println("My id is "+id);
                        sleep(1000);
                        j++;
                }
        } catch (InterruptedException e) {
        }
}
```

- To check a thread's interrupt status — `t.isInterrupted()` : does not clear flag
- Can give up running status
  - `yield()` gives up active state to another thread
  - Static method in `Thread`
  - Cooperative scheduling — thread loses control only if it yields
- Waiting for other threads
  - `t.join()` waits for `t` to terminate

Concurrent Programming: An Example

- Exercise
  - narrow bridge; accommodates traffic in only one direction at a time
  - when a car arrives at the bridge:
    - can cross **if** cars on the bridge going in the same direction
    - can cross **if** no other car on the bridge
    - wait for the bridge **if** cars on the bridge going in the opposite direction
- Design a class `Bridge` to implement consistent one-way access for cars
  - `Bridge` has a public method `public void cross (int id, boolean d, int s)`
    - `id` : identity of the car
    - `d` : `true` is North and `false` is South
    - `s` : indicates time taken to cross (milliseconds)
- The "data" that is shares is the `Bridge`
  - Number of cars on bridge — `int bcount`
  - Current direction of bridge — `boolean direction`
- The method `cross` changes the state of the bridge
  - Concurrent execution of `cross` can cause problems
  - make `cross` a synchronized method is too restrictive (only one car will be able to cross the bridge at a time)
- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print diagnostics
- Affecting the state:
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars
  - make `enter` and `leave` synchronized
- Code for `cross`

```
public void cross (int id, boolean d, int s) {

        // get on the bridge if you can
        enter(id, d);
```

```
        // takes time to cross the bridge
        try{
                thread.sleep(s);
        } catch(InterruptedException s) {}

        // get off the bridge
        leave(id);
}
```

- Code for `enter`

```
private synchronized void enter (int id, boolean d) {
        Date date;

        // while there are cars going in the wrong direction
        while (d != direction && bcount > 0){
                date = new Date();
                System.out.println("Car "+id+" going "+
                                                        direction_name(d)+" stuck at
"+date);

                // wait for turn
                try{
                        wait();
                } catch (InterruptedException e){}
        }

        if (d != direction) {
                direction = d;
                date = new Date();
                System.out.println("Car "+id+" switches bridge direction to "
                                                        +direction_name(direction)+" at
"+date);
        }

        bcount++;

        date = new Date();
        System.out.println("Car "+id+" going "+direction_name(d)
                                                +" enters the bridge at "+date);
}
```

- Code for `leave`

```
private synchronized void leave(int id){
        Date date = new Date();
        System.out.println("Car "+id+" leaves at "+date);

        bcount--;

        if (bcount==0){
                notifyAll();
        }
}
```

- **Thread Safe Collection**
  - Thread safety guarantees consistency of individual updates.

```
public class bank_account{
        double accounts[100];
```

```
        public boolean transfer(double amt,
                                                int source,
                                                int target){
                while (accounts[source] < amt) { wait();}
                accounts[source] -= amt;
                accounts[target] += amt;
                notifyAll();
                return true;
        }

        public double audit() {
                double bbalance = 0.0;
                for (int i=0; i<100; i++){
                        balance += accounts[i];
                }
                return balance;
        }

        public double current_balance(int i) {
                return accounts[i];
        }
}
```

- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner
- Formally, linearizability. Contrast with serializability in databases, where transactions (sequences of updates) appear atomic
- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restricted access to nodes on either side of insert/delete
- built-int collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - low-level locking is done automatically to ensure consistent local updates
  - Sequence of updates (transfer from one account to another) still need to be manually synchronized to work properly
- Producer-Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them
- Bank account example
  - transfer threads insert transfer instruction into shared queue
  - update thread process instruction from the queue, modifies bank accounts
  - no synchronization necessary
- Blocking queues block when
  - you try to add an element when the queue is full
  - you try to remove an element when the queue is empty
- Blocking automatically balances the workload
  - Producers wait if consumer are slow and the queue fills up
  - Consumers wait if producers are slow to provide items to process

Week 12

- **Graphical Interfaces and Event-Driven Programming**
  - In parallel to main activity, record and respond to events (user interactions)
    - Web browser renders current page

    - Clicking on a link loads a different page
- Keeping track of events
    - remember coordinates and extent of each window
    - track coordinates of mouse
    - OS reports mouse click at (x, y)
        - check which windows are positioned at (x, y)
    - Tedious and error-prone
    - Programming language support for higher level events
        - Run time support for language maps low level events to high level events
- Better Programming Language support for events
    - Programmer directly defines `components` such as windows, buttons, ... that generate high level events
    - Each event is associated with a `listener` that know what to do
        - e.g., clicking `Close window` exits application
    - Setting up an association between components and listeners
- Example: A `Button` with one event, press button

```
interface ButtonListener {
        public abstract void buttonpush(..);
}

class Myclass implements ButtonListener{
        ...
        public void buttonpush(...){
                // what to do when button is pushed
        }
}

Button b = new Button();
Myclass m = new Myclass();
b.add_listener(m); // Tell b to nofity m when pushed
```

- `Timer`
    - `Myclass m` creates a `Timer t` that runs in parallel
    - `Timer t` notifies `Timerowner` when it is done, via function `timerdone()`
    - timer duration elapsing is an event, and `Timerowner` is notifies when the event occurs

Swing Toolkit

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- One listener can listen to multiple objects
- One component can inform multiple listeners
- `JButton` : Swing class for buttons
    - Corresponding listener class is `ActionListener`
    - invokes `actionPerformed(...)` in listener
    - Button push is an `ActionEvent`

```
public class MyButtons{
        private JButton b;
        public MyButtons(ActionListener a) {
                b = new JButton("MyButton"); // Set the label on the button
                b.addActionListener(a); // Associate a listener
        }
}

public class MyListener implements ActionListener{
        public void actionPerformed(ActionEvent e) {
                // What to do when a button is pressed
        }
```

```
}

public class XYZ {
        MyListener l = new MyListener();
        MyButtons m = new MyButtons(l); // Button m, reports to l
}
```

- Embedding the button in a panel
  - `JPanel` will also serve as the event listener

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel implements ActionListener {
        private JButton redButton;

        public ButtonPanel() {
                redButton = new JButton("Red");
                redButton.addActionListener(this);
                add(redButton);
        }

        public void actionPerformed(ActionEvent evt) {
                Color color = Color.red;
                setBackground(color);
                repaint();
        }
}
```

- Embed the panel in a frame
  - `JFrame`
  - Items to be displayed have to be added to `ContentPane`

```
public class ButtonFrame extends JFrame implements WindowListener {
        private Container contentPane;

        public ButtonFrame(){
                setTitle("ButtonTest");
                setSize(300,200);

                // ButtonFrame listens to itself
                addWindowListener(this);

                // ButtonPanel is added to the contentPane
                contentPane = this.getContentPane();
                contentPane.add(new ButtonPanel());
        }

        // Implement WindowListener
        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }
        // Other six types of events can be ignored
}


public class ButtonTest{
        public static void main(String[] args) {
                EventQueue.invokeLater(
                        () -> {
                                JFrame frame = new ButtonFrame();
                                frame.setVisible(true);
```

```
                        }
                );
        }
}
```

- **More Swing Examples**
  - One listener can listen to multiple objects

```
public class ButtonPanel extends JPanel implements ActionListener {
        private JButton yellowButton, blueButton, redButton;

        public ButtonPanel(){
                yellowButton = new JButton("Yellow");
                blueButton = new JButton("Blue");
                redButton = new JButton("Red");

                // Make the panel listen to all three buttons
                yellowButton.addActionListener(this);
                blueButton.addActionListener(this);
                redButton.addActionListener(this);
                add(yellowButton);
                add(blueButton);
                add(redButton);
        }

        public void actionPerformed(ActionEvent evt){
                Object source = evt.getSource();
                Color color = getBackground();

                if (source == yellowButton) {
                        color = Color.yellow;
                } else if (source == blueButton) {
                        color = Color.blue;
                } else if (source == redButton) {
                        color = Color.red;
                }

                setBackground(color);
                repaint();
        }
}
```

  - Multicasting: Multiple listeners for an event
    - Two panels each with three buttons, `Red` `Blue` `Yellow`
    - Clicking a button in either panel should change the background colour in both panels

```
import...
public class ButtonPanel extends JPanel implements ActionListener {
        private JButton yellowButton, blueButton, redButton;

        public ButtonPanel(){
                ...
                yellowButton.setActionCommand("YELLOW");
                blueButton.setActionCommand("BLUE");
                redButton.setActionCommand("RED");

                add(yellowButton);
                add(blueButton);
                add(redButton);
        }

        public void actionPerformed(ActionEvent evt){
                Color color = getBackground();
```

```
                String cmd = evt.getActionCommand();

                if (cmd.equals("YELLOW")) {
                        color = Color.yellow;
                } else if (cmd.equals("BLUE")) {
                        color = Color.blue;
                } else if (cmd.equals("RED")) {
                        color = Color.red;
                }

                setBackground(color);
                repaint();
        }

        public void addListener(ActionListener o) {
                // Add a common listener for all buttons in this panel
                yellowButton.addActionListener(o);
                blueButton.addActionListener(o);
                redButton.addActionListener(o);
        }
}


public class ButtonFrame extends JFrame implements WindowListener {
        private Container contentPane;
        private ButtonPanel b1, b2;

        public ButtonFrame() {
                ...
                b1 = new ButtonPanel();
                b2 = new ButtonPanel();

                // Each panel listens to both sets of buttons
                b1.addListener(b1); b1.addListener(b2);
                b2.addListener(b1); b2.addListener(b2);

                contentPane = this.getContentPane();
                contentPane.setLayout(new BorderLayout());
                contentPane.add(b1, "North");
                contentPane.add(b2, "South");
        }
}
```

- Checkboxes
  - `JCheckbox` : a box that can be ticked

```
import ...
public class CheckBoxPanel extends JPanel implements ActionListener {
        private JCheckBox redBox;
        private JCheckBox blueBox;

        public CheckBoxPanel(){
                redBox = new JCheckBox("Red");
                blueBox = new JCheckBox("Blue");

                redBox.addActionListener(this);
                blueBox.addActionListener(this);

                redBox.setSelected(false);
                blueBox.setSelected(false);

                add(redBox);
                add(blueBox);
        }
```

```
        public void actionPerformed(ActionEvent evt) {
                Color color = getBackground();

                if (blueBox.isSelected()){
                        color = Color.blue;
                }
                if (redBox.isSelected()){
                        color = Color.red;
                }
                if (blueBox.isSelected() && redBox.isSelected()){
                        color = Color.green;
                }

                setBackground(color);
                repaint();
        }
}
```