1. **Enum**

Link: https://www.csee.umbc.edu/courses/undergraduate/202/spring12/lectures/enums.pdf

+ Java Platform SE 7
- Definition: Enumerated values are used to represent a set of named values (enum is object)

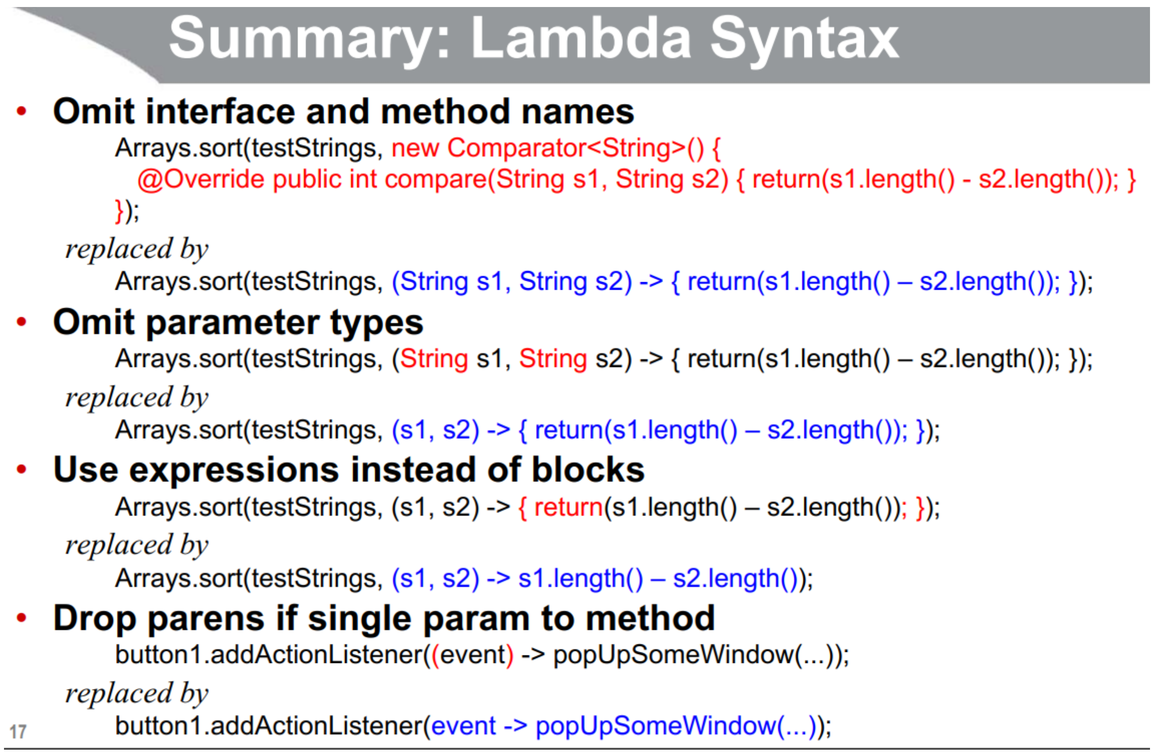
- Benefit of enum:
+ Acceptable values are obvious -> must choose one of the enumerated values defined already
+ Type safety -> compiler check type of enum
+ Name-spacing -> every value is name-spaced off of the enum type itself
+ Printable
+ Storage of additional information
+ Retrieve of all enumerated values as an array -> Suit[] suits = Suit.values();
+ Comparison of Enumerated values -> if(suit == Suit.CLUBs)

2. **Lambda**

Link: http://www.coreservlets.com/java-8-tutorial/

+ Java Platform SE 8
- Lambda expression is object
Lambda1:
- Advantage of Lambda:
+ Concise syntax (ngan ngon)
+ Deficiencies with anonymous inner classes (inner class:bulky, hard to optimize)
+ Convenient for new streams library, support streams
+ Programmers are used to the approach
+ Encourage functional programming

- DisAdvantage of lambda:
+ Type of a lambda is class that implements interface, not a “real” function
    • Must create or find interface first, must know method name
+ Cannot use mutable local variables



Summary: Lambda Syntax

- **Omit interface and method names**

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override public int compare(String s1, String s2) { return(s1.length() - s2.length()); }
});
```

replaced by

```
Arrays.sort(testStrings, (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

- **Omit parameter types**

```
Arrays.sort(testStrings, (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

replaced by

```
Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() – s2.length()); });
```

- **Use expressions instead of blocks**

```
Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() – s2.length()); });
```

replaced by

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() – s2.length());
```

- **Drop parens if single param to method**

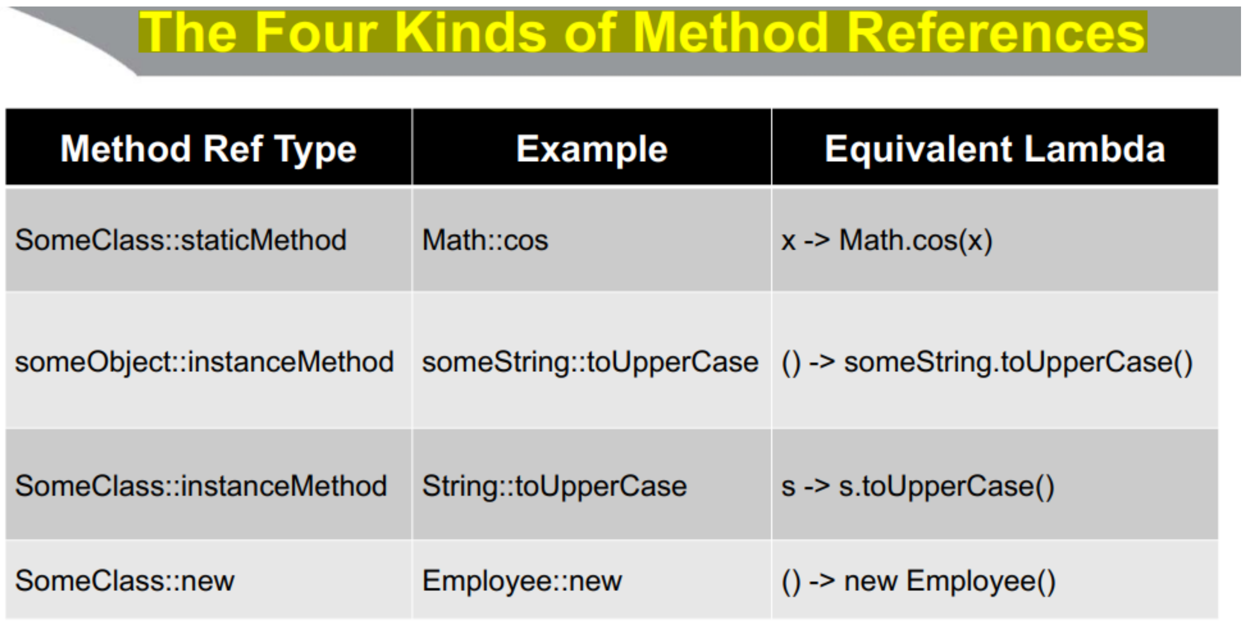
```
button1.addActionListener((event) -> popUpSomeWindow(...));
```

replaced by

```
button1.addActionListener(event -> popUpSomeWindow(...));
```

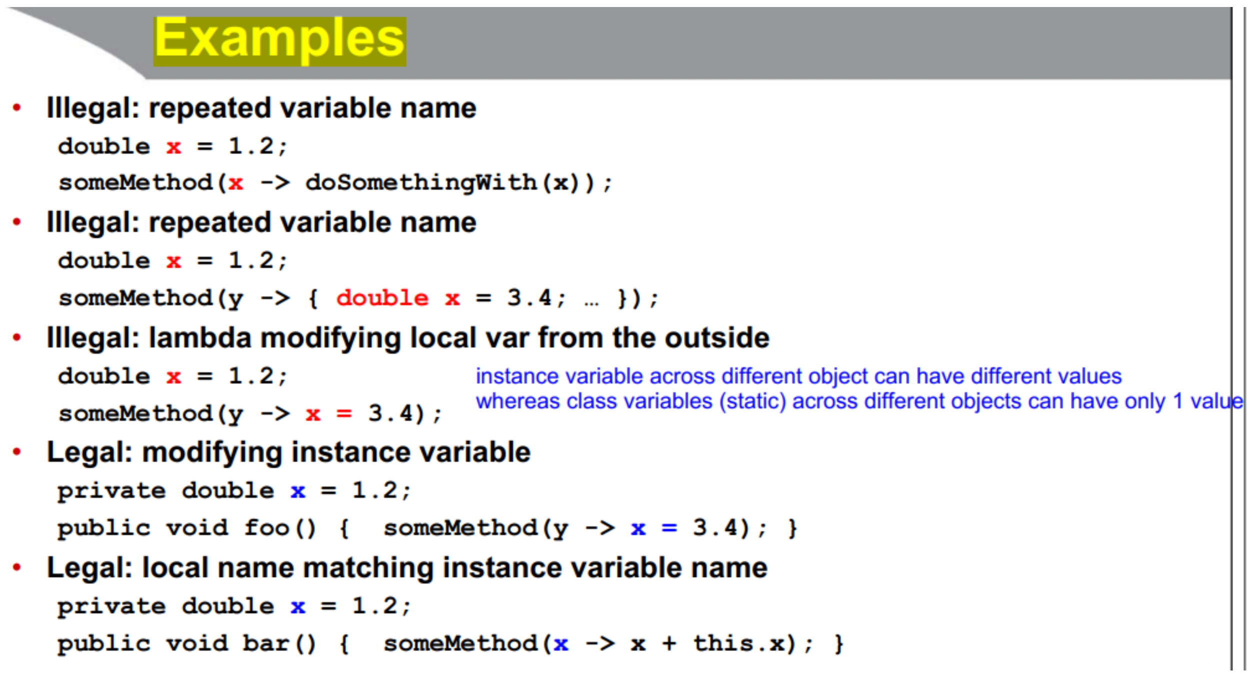
17

# Lambda2:
@FunctionalInterface
+ Functional Interfaces in Java 8 allows exactly one abstract method inside them.
+ @FunctionalInterface annotation is useful for compilation time checking of your code.
+ This feature in Java, which helps to achieve functional programming approach.



The Four Kinds of Method References

Method Ref Type	Example	Equivalent Lambda
SomeClass::staticMethod	Math::cos	x -> Math.cos(x)
someObject::instanceMethod	someString::toUpperCase	() -> someString.toUpperCase()
SomeClass::instanceMethod	String::toUpperCase	s -> s.toUpperCase()
SomeClass::new	Employee::new	() -> new Employee()



Examples

- **Illegal: repeated variable name**

```
double x = 1.2;
someMethod(x -> doSomethingWith(x));
```

- **Illegal: repeated variable name**

```
double x = 1.2;
someMethod(y -> { double x = 3.4; … });
```

- **Illegal: lambda modifying local var from the outside**

```
double x = 1.2;
someMethod(y -> x = 3.4);
```

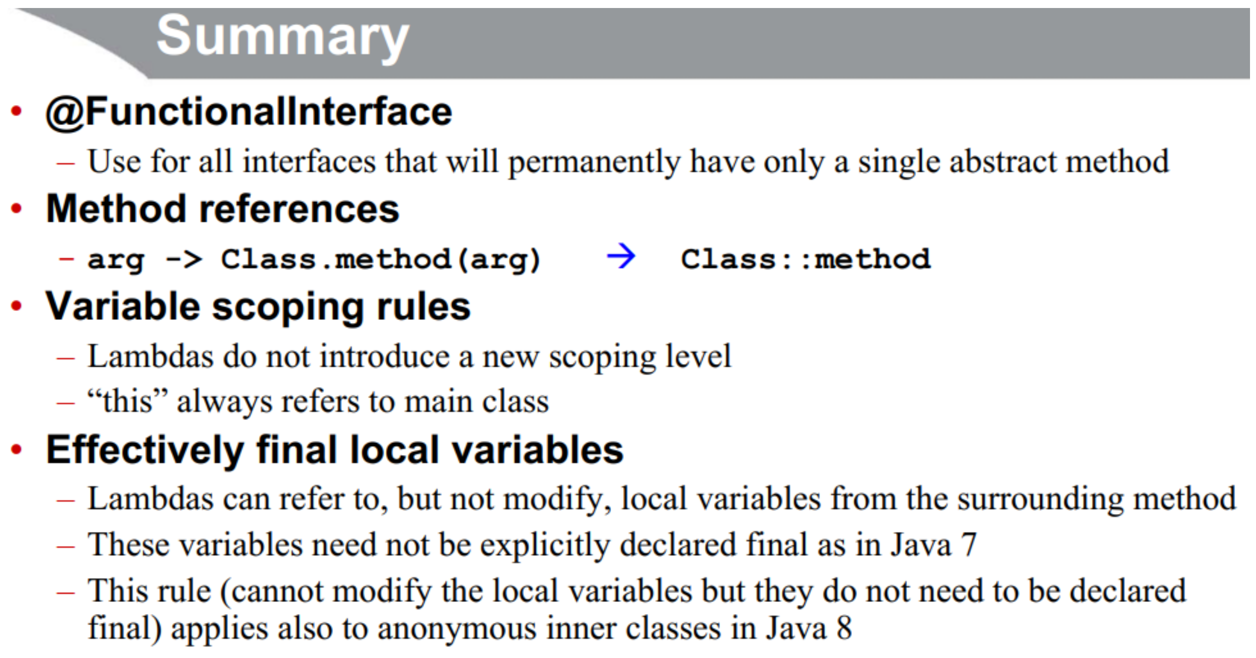
instance variable across different object can have different values
whereas class variables (static) across different objects can have only 1 value

- **Legal: modifying instance variable**

```
private double x = 1.2;
public void foo() {  someMethod(y -> x = 3.4); }
```

- **Legal: local name matching instance variable name**

```
private double x = 1.2;
public void bar() {  someMethod(x -> x + this.x); }
```



Summary

- **@FunctionalInterface**
  - Use for all interfaces that will permanently have only a single abstract method
- **Method references**
  - `arg -> Class.method(arg)` → `Class::method`
- **Variable scoping rules**
  - Lambdas do not introduce a new scoping level
  - "this" always refers to main class
- **Effectively final local variables**
  - Lambdas can refer to, but not modify, local variables from the surrounding method
  - These variables need not be explicitly declared final as in Java 7
  - This rule (cannot modify the local variables but they do not need to be declared final) applies also to anonymous inner classes in Java 8

# Lambda3:

- Runnable: Ø → Ø
- Consumer: T → Ø (Lets you make a "function" that takes in a T and no return value)
- IntConsumer: int → Ø
- Supplier: Ø → T (Lets you make a no-arg "function" that returns a T)
- Function: T_1 → T_2 (Lets you make a "functions" that takes in a T1 and returns a T2)
- IntFunction: int → T

- IntUnaryOperator: int → int
- BiFunction: $(T_1, T_2) \rightarrow T_3$ (Lets you make a “functions” that takes 2 arguments T1,T2 and return T3)
- IntBinaryOperator: (int, int) → int
- Predicate: T → Boolean (lets you make a “function” to test a condition)
- BinaryOperator: (T1,T2) -> T3 ~ BiFunction<T,U,R> where T, U, R are all the same type

3. **Stream**

Link: http://www.coreservlets.com/java-8-tutorial/

- Java 8
- Making streams more powerful, faster, and more memory efficient than Lists
- The three coolest properties:
 • Lazy evaluation • Automatic parallelization • Infinite (unbounded) streams

#Stream1:
- 3 ways to make a Stream:
+ From Lists: List<String> words =…; words.stream().map()…
+ From object arrays: Employee[] workers =…; Stream.of(workers).map()…
+From individual elements: Employee[] e1 =…; Employee[] e2 =…; Employee[] e3 =…; Stream.of(e1,e2,….).map()…
- findFirst(): return Optional<T> -> ~ check the optional is empty?
- Turning Streams into Pre-Java-8 Data Structures -> do this only at the end, after you have done all the stream.
+ Output as a list:
List<String> w = someStream**.collect(Collectors.toList())**
List<Employee> w = someStream**.collect(Collectors.toList())**:
+ Output as an array:
String[] w = someStream.**toArray(String[]::new)**
Employee[] w = someStream.**toArray(Employee[]::new)**

#Stream2:
- **limit**(n) returns a Stream of the first n elements.
- **skip**(n) returns a Stream starting with element n (i.e., it throws away the first n elements)
- **reduce**(starterValue, binaryOperator): ~ reduce(baseValue,Integer:sum())

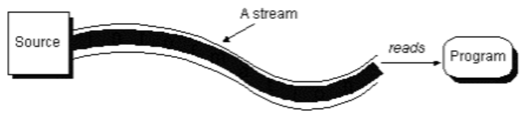
4. **Unit Test**

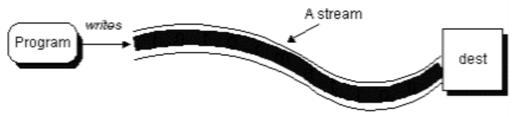
- JUnit 5 requires Java 8 (or higher) at runtime
- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
+ The JUnit Platform serves as a foundation for launching testing frameworks on the **JVM**
+ JUnit Vintage provides a TestEngine
+ JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5

5. Serialization

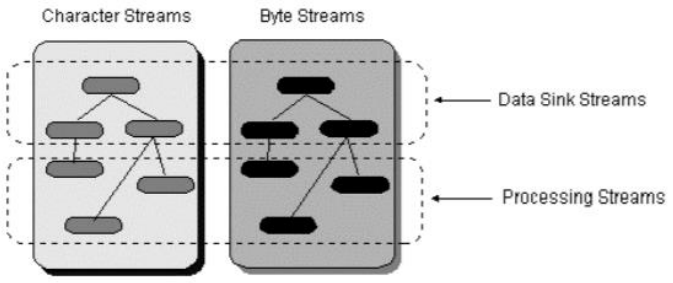
Serialization to transform binary form to binary data
Deserializable to convert data to binary (object)
Write to a file and read back to an object

- **Stream**:
+ A program reads/writes information from/to a channel. In Java, a channel from where a program may read or write information is referred to as a STREAM.

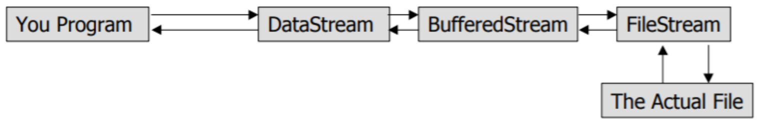




+ There are two kinds of Streams: Bytes Streams (classes named *Stream); Character Streams (classes named *Reader or *Writer)
+ **Data sink streams**: connected directly with the source or destination. (the end of the stream)
+ **Processing streams**: connected to other streams to provide further processing transparently (filtering, compression, etc..)



+ A **stream chain** is: a chain of processing streams. one sink stream.

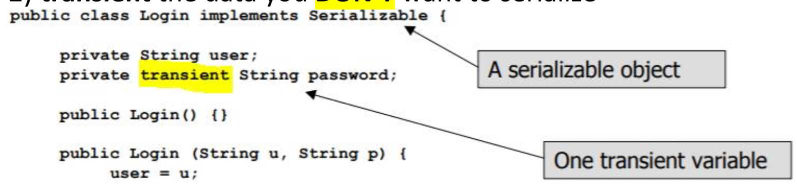


Description: When reading (using the write method): Your program asks the data stream to read a real number. The data streams asks the buffer stream to read a number of bytes corresponding to the length of a real number. The buffer stream asks the file stream to read some more bytes so that they are buffered for the next read. The file stream actually reads the bytes The data is passed back and interpreted by each stream

- **Persistence** means having an object's life independent from the life time of the application in which it is running. One way to implement persistence is storing objects and then retrieving them.

- **Serialization**:
+ **The process of sending an object through a stream is referred to as SERIALIZATION.**
+ The ObjectStream classes implement serialization and deserialization of objects.

+ There are two classes implementing processing streams: **ObjectInputStream**, **ObjectOutputStream**
+ Security in Serialization: 2 ways
1) **transient** the data you **DON'T** want to serialize



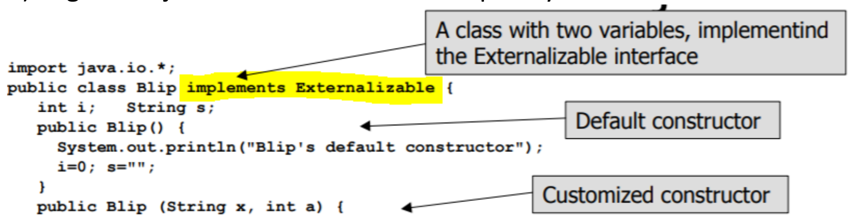
```
public class Login implements Serializable {

    private String user;
    private transient String password;

    public Login() {}

    public Login (String u, String p) {
        user = u;
```

2) Tag the object as Externalizable to explicitly declare the data we want to serialize



```
import java.io.*;
public class Blip implements Externalizable {
   int i;   String s;
   public Blip() {
     System.out.println("Blip's default constructor");
     i=0; s="";
   }
   public Blip (String x, int a) {
```

A class with two variables, implementind the Externalizable interface
Default constructor
Customized constructor

- Summary:
+ We have seen the Java STREAM mechanism and class organization.
+ We have designed and build Stream Chains.
+ We have made our own Streams and included them in other Stream Chains.
+ We have Serialized objects.
+ We have controlled the serialization process through the transient and externalizable mechanisms (security)

6. Network-socket

Link: https://www.slideshare.net/tusharkute/network-programming-in-java
Link: https://slideplayer.com/slide/5150902/

Client <-> Network <-> Server

Socket
- Sockets provide an interface for programming networks at the transport layer-> Network communication using Socket as I/O.
- Socket is endpoint for communication between two machines.
- Socket-based communication can communicate on program in Java or Non-Java.
- Socket uses TCP to communicate over the network

Constructor:
Socket(String remoteHost, int remotePort)
Socket(InetAddress ip, int remotePort)

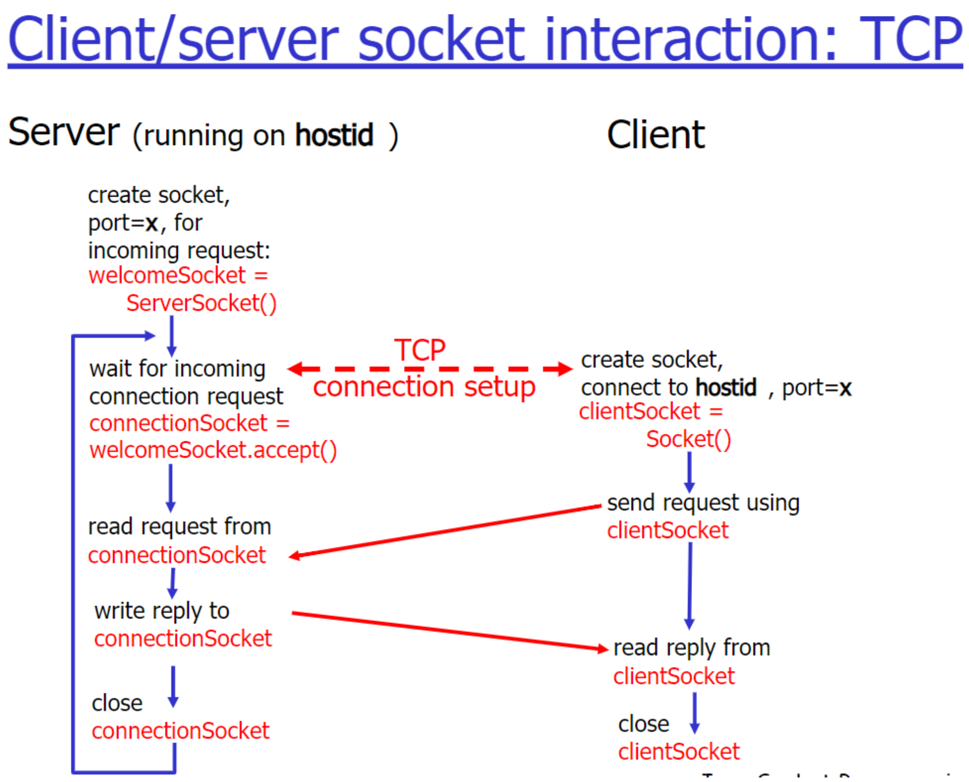
TCP-Transmission Control Protocol
- TCP provides a reliable flow of data between 2 computers (point-to-point).
Ex: HTTP,FTP, Telnet require a reliable communication channel.
- The *URL,URLConnection, Socket, ServerSocket* classes all use Transmission Control Protocol (TCP) to communicate over the network

UDP-User Datagram Protocol

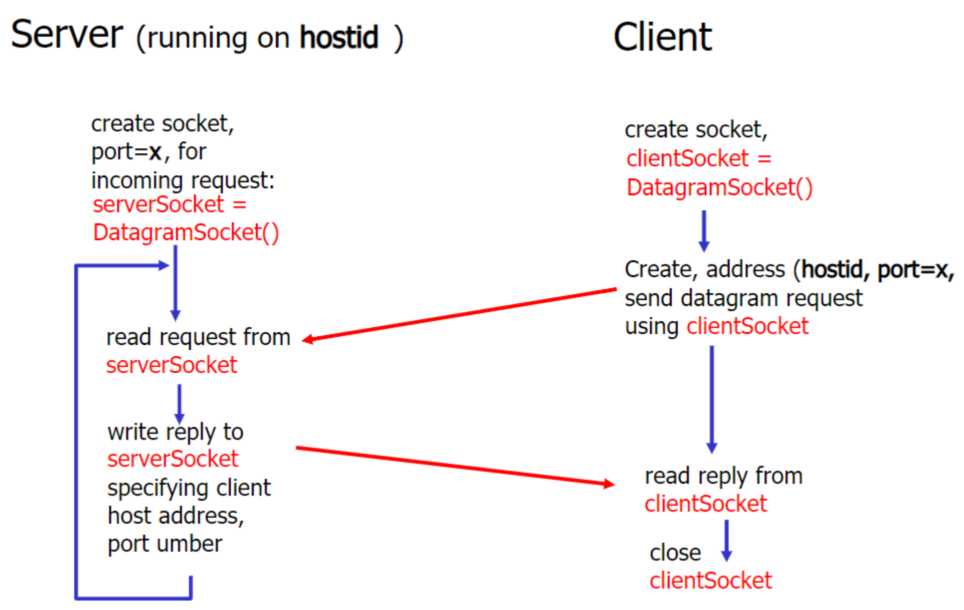
- UDP sends independent packets of data (called datagrams) from one computer to another with no guarantees about arrival -> not reliable, but good in speed and cost

Ex: streaming media, games, Internet telephony…

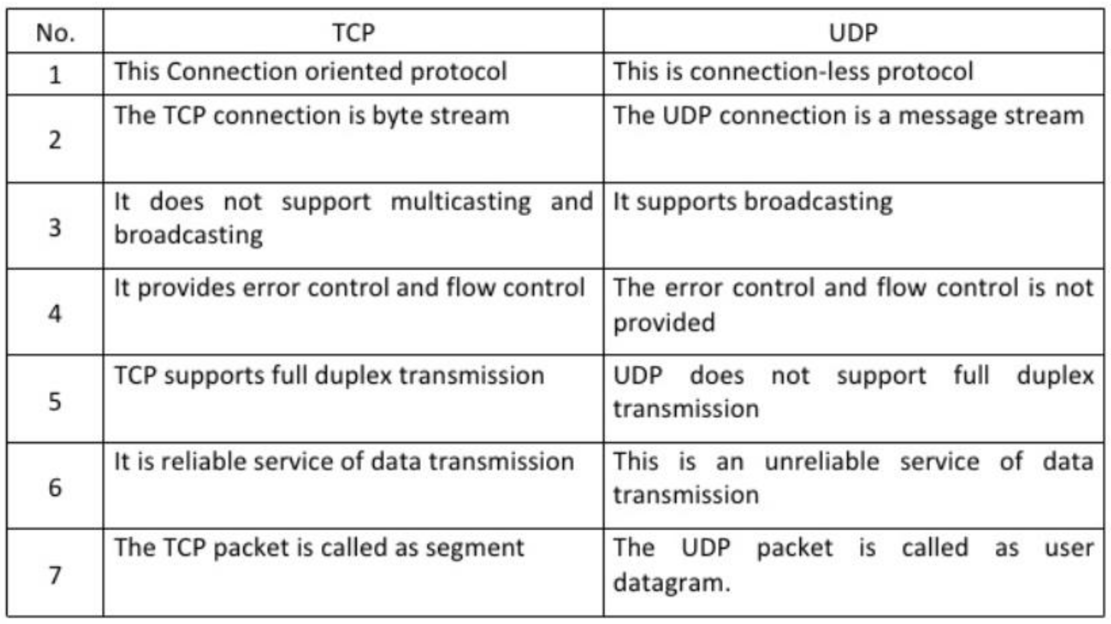
- The *DatagramPacket, DatagramSocket, MulticastSocket* classes are for use with User Datagram Protocol (UDP)


Client/server socket interaction: TCP
Server (running on hostid)
Client
create socket, port=x, for incoming request:
welcomeSocket = ServerSocket()
wait for incoming connection request
connectionSocket = welcomeSocket.accept()
TCP connection setup
create socket, connect to hostid , port=x
clientSocket = Socket()
send request using clientSocket
read request from connectionSocket
write reply to connectionSocket
read reply from clientSocket
close connectionSocket
close clientSocket

Client/server socket interaction: UDP


Server (running on hostid)
Client
create socket, port=x, for incoming request: serverSocket = DatagramSocket()
create socket, clientSocket = DatagramSocket()
Create, address (hostid, port=x, send datagram request using clientSocket
read request from serverSocket
write reply to serverSocket specifying client host address, port umber
read reply from clientSocket
close clientSocket

TCP vs. UDP



No.	TCP	UDP
1	This Connection oriented protocol	This is connection-less protocol
2	The TCP connection is byte stream	The UDP connection is a message stream
3	It does not support multicasting and broadcasting	It supports broadcasting
4	It provides error control and flow control	The error control and flow control is not provided
5	TCP supports full duplex transmission	UDP does not support full duplex transmission
6	It is reliable service of data transmission	This is an unreliable service of data transmission
7	The TCP packet is called as segment	The UDP packet is called as user datagram.

Port # IP

Port:

- TCP and UDP use Ports to deliver the data to the right application
- 16 bit integer value (2^16), 0 – 1023 (well-known ports) to 65535
- FTP (20,21); TELNET (23); SMTP (25); POP3 (110); HTTP (80); DSN(53)

IP:
- IP as address 32bit
- IP uses to deliver data to the right computer on the network
- Java.net.InetAddress-> both IP address and domain name

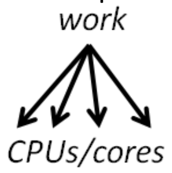
7. Thread

Link: 27-concurrency.ppt

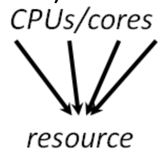
- **Thread safety**: Able to be used **concurrently** by multiple threads. (run at the same time)
+ Many of the Java library classes are *NOT* thread safe! Ex: ArrayList, java.util, StringBuilder, Java GUIs…
+ But Random, System.out are thread safe

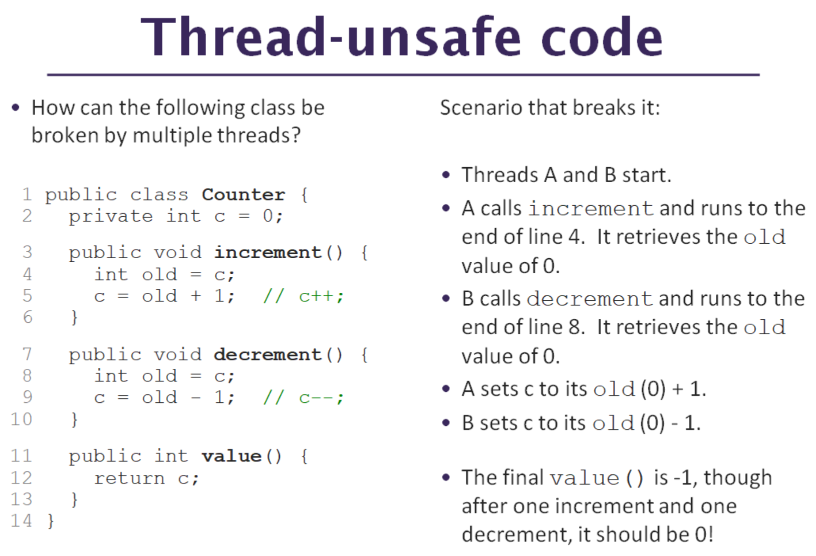
- **Time slicing:** If a given piece of code is run by two threads at once:
+ The **order** of which thread gets to run first is **unpredictable**.
+ How many **statements** of one thread run before the other thread runs some of its own statements is **unpredictable**.

- **Parallel:** Using multiple processing resources (CPUs, cores) at once to solve a problem faster.
+ Example: A sorting algorithm that has several threads each sort part of the array.



- **Concurrent:** Multiple execution flows (e.g. threads) accessing a shared resource at the same time.
+ Example: Many threads trying to make changes to the same data structure (a global list, map, etc.).





Thread-unsafe code

- How can the following class be broken by multiple threads?

```
 1 public class Counter {
 2   private int c = 0;
 3   public void increment() {
 4     int old = c;
 5     c = old + 1;  // c++;
 6   }
 7   public void decrement() {
 8     int old = c;
 9     c = old - 1;  // c--;
10   }
11   public int value() {
12     return c;
13   }
14 }
```

Scenario that breaks it:

- Threads A and B start.
- A calls `increment` and runs to the end of line 4. It retrieves the `old` value of 0.
- B calls `decrement` and runs to the end of line 8. It retrieves the `old` value of 0.
- A sets c to its `old` (0) + 1.
- B sets c to its `old` (0) - 1.
- The final `value()` is -1, though after one increment and one decrement, it should be 0!

- **synchronized lock**: Every Java object can act as a "lock" for concurrency
synchronized (object) {
  statement(s);
}

- A **synchronized method** grabs the object or class's lock at the start, runs to completion, then releases the lock.
Ex:
// synchronized method: locks on "this" object
public **synchronized type name**(**parameters**) { ... }
// synchronized static method: locks on the given *class*
public static **synchronized type name**(**parameters**) { ... }

- **Critical section**: A piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread.

- **Volatile field**: An indication to the VM that multiple threads may try to access/update the field's value at the same time.
- Terms:
+ **liveness**: Ability for a multithreaded program to run promptly.
+ **deadlock**: Situation where two or more threads are blocked forever, waiting for each other.
+ **livelock**: Situation where two or more threads are caught in an infinite cycle of responding to each other
+ **starvation**: Situation where one or more threads are unable to make progress because of another "greedy" thread.