Java Basics Call super-class constructors by making super(..) call
Arrays are covariant, so assignment may fail at runtime
Use "package" for namespace
"import" to bring in package
Default access modifier allows access by same class, package
Protected variables are accessible in the package they are defined in and within subclasses but only when accessed on instances of the subclass (i.e. super-class instances are not allowed)

Modifiers "final" fields must be initialized by every code path that creates an instance
"strictfp" on classes, methods
"volatile" causes caching to not be permitted
"synchronized" locks on the class or object instance
"transient" on fields
"native" methods are implemented in native code

Nested Classes Inner classes, static nested classes, nested interfaces, anonymous inner classes
Can access enclosing classes using EnclosingClass.this

MVC Model(whatever), View(update), Controller(Model, View[], changeModel, addView)
Potential for cascading updates and computational overhead

Singleton Singleton(static getInstance)

Factory abstract Factory(build), MyFactory() : Factory
Ensures e.g. no MotifWindow with MacScrollBar (i.e. consistency)

Adapter Implement target interface in terms of adaptee operations
Can use the adapter with any subclass of the adaptee

Visitor Node(acceptVisitor), Visitor(apply)
The node applies the visitor to its data and then make its children accept the visitor
Methods implementing a particular operation are kept together in a sub-class of Visitor

UML Triangle generalisation arrows point towards super-classes
Normal aggregation arrows point away from the aggregator
Dashed relationship arrows point from parent to child (e.g. MacFactory -> MacScrollBar)

Reflection getClass(), getMethods(), getInterfaces(), getSuperclass(), newInstance(): 0-ary ctor used
Security manager "checkMemberAccess" must permit general access for Member.PUBLIC and "checkPackageAccess" must permit in-package reflection

Serialization Object{Input/Output}Stream
"Serializable" marker interface
Can also implement readObject(ObjectOutputStream), writeObject(ObjectInputStream)
readResolve/writeReplace let you replace the object wholesale with a new one (poss. Implementing {read/write}Object itself!)
"Externalizable" interface lets you implement readExternal and writeExternal to *totally* serialize the object state: super-classes are not done by default! Also, the class is constructed before deserialization. This can be efficient in speed (no reflection) and size (total control)

GUIs AWT: LCD, native controls
Swing: common, native theme
Events: AWTEvent, Listener and Adapter pattern for addXListener

Garbage Collection Increased run time performance since cleanup occurs when idle
Stability for long running apps
No need to agree which module has deallocation responsibility
Programmer never thinks about memory allocation in the app!
Less control over footprint
More threads at run time?
Object reachable if referenced or still needs to be finalized
References can be restored by a finalizer, but will only be run once, and there is no control over their thread or execution order
{Soft, Weak, Phantom}Reference

can be associated with a queue
Soft references MAY be cleared by a GC if the object dies (depends on timestamp etc)
Weak references WILL be cleared by a GC if the object dies
Phantom references are enqueued after finalisation has run and can be used for cleaning up native resources (via subclass)

Class Loaders
Test whether class is loaded, delegate to parent class loader and then findClass to actually load the bytes to defineClass
Types are identified by <Loader, Fully Qualified Name>: this can be used to host separate applications within on JVM!

Threading
Only get InterruptedException on blocking calls such as sleep, wait
Otherwise check for interrupt using Thread.isInterrupted()
Can explicitly yield

Problems
Liveness: deadlock, livelock, starvation, missed wake-up
Field accesses are atomic only to 32 bits (long, double = danger)
Priority inversion can be alleviated with *priority inheritance* where locks have a priority = the maximum priority of a requestor that the holder of the lock is boosted up to, now get push-through blocking of other threads by the boosted thread!

Deadlock
Requirements: resource request can be refused, resources are held while waiting, no resource preemption allowed, circular wait
Detect by looking for cycles in object allocation graphs (in which you have thread and resources, arrows between the two sets show requests and holdings)
Given allocations $A_{i,j}$ and requests $R_{i,j}$ thread i and resource j with working vector W of currently available objects

1. Find an unmarked thread whose requests R_i can be met. If one does not exist, terminate
2. Set $W = W + A_i$, mark the thread and repeat

If some thread is unmarked then deadlock has occurred

Condition Variables
wait(), notify(), notifyAll(): must hold the associated lock

Locking
Implement by means of a "locking protocol" class
MRSW: reader count,FCFS: ticket
CAS/TAS w/ resume queue good
Semaphores: P decrements and blocks if the result is < 0, V increments and if the result is <= unblock a blocked thread
Event count: has event number, increased by advance(), got by read() and has await(i)
Sequencer: supports only ticket() which increments value by one and returns the old value
Monitor: ADT where mutual exclusion is enforced between invocations of its operations
Active Object: has conceptual mutual exclusion by performing operations on a dedicated thread

Distributed Systems
Problems: parallel execution, communication delayed, independent failure, no clock

Naming
Identify resources to access via late binding of names
Unique IDs: allocation easy but centralised. What about reuse?
Hierarchical namespace: local allocation following real world control delegation, has locality of access, but lookups complex
Pure name: contains no information about the object
Impure names: typically prevent the object from moving/changing
Name service can be enhanced with caching by clients/servers, replication, distribution

Sockets
UDP: loss, duplication, reordering but checksum and framing OK. Need to implement flow, congestion control, loss recovery
TCP: reliable, bidirectional, flow and congestion control OK. Need to implement framing, marshalling, 1-* communication
Java: Datagram{Socket, Packet},

ServerSocket, Socket

RMI

Server registers a reference to a remote object with the registry (a name service) and deposits associated .class files in a shared location (the RMI codebase)
Client queries the registry for a reference to the remote object and grabs code from the codebase if it is not locally available, then makes RMI calls
Requires own security and re-registering at the application level
Parameters/results typically deep-copied over RMI, but objects implementing Remote are passed by reference
Our own remotable objects must extend Remote somehow, and such an interface must mark all methods with RemoteException
RMI creates one thread per incoming connection (prevents deadlock), then emulates locks when re-entrant calls are made
Retry semantics after timeout: at-most-once or "exactly" once (retry with same RPC id)..?

Transactions

Atomicity, consistency, isolation, durability: ACID test
Serializability: a concurrent execution that gives the same result as some serial execution
Can represent this as a "history graph": a edge from a to b means that a happened before b (w/ transitivity). "Cycles" indicate non-serializable execution orders
Lost updates, dirty reads (before a commit), unrepeatable reads
Isolation: strict and non-strict (more concurrency but can have delays on commit waiting for transactions it dirty read from to commit, and cascading abort)

2PL

Acquire, release, do other operations during both phases
Can use application knowledge
Ensures serializable execution
Deadlock free with locking order
Can be complex to use
Has non-strict problems, use strict variant to solve (hold all locks until after commit)
Performance may be bad due to lock overhead and restrictiveness

TSO

Each transaction has a timestamp (e.g. start time). The timestamp will give a serializable order for the transactions (if two transactions access the same object they must do so according to their timestamp order)
Give each object a timestamp field, when it is accessed by a transaction check the timestamp: if transaction is later update the object else abort transaction
Abort decision is made with local information and simple to do
No locks may increase concurrency and is deadlock free
Requires rollback, some serializable orders are rejected, and cascading aborts are still possible if lower T aborts

OCC

Assumes that concurrent transactions rarely conflict and so only check serializability at commit time, using shadow copies so no cascades/slow abort
Assign start time timestamp to transaction: that of the last committed transaction. When taking shadow copies record the timestamp of the most recent transaction to update that object (stored in another table). When validating compare each shadows timestamp against the start time and if later abort. Then check against all transactions in the list after the start time with the changes the current transaction made: If we see a conflict then abort (this prevents lost updates).

Logging

Record details of updates to do in the log in (transaction, old, new) form with transaction control signals (start, abort, commit) as well
Only write to actual memory once we are sure we have the change logged as well
Make checkpoints where log

records are forced out to non-volatile storage: will now only consider transactions ending after the checkpoint on crash, have to REDO transactions that had committed before crash and UNDO those that were in progress before the crash
Can also implement this by allocating shadow objects in other parts of memory and then atomically flipping a pointer: dead objects could reclaim lazily

Generics

Parameter types cannot be primitives or arrays
Type erasure ☹
Static fields, methods are shared between generic instances: thus cannot refer to the type parameter in e.g. initializers
Wildcards let you operate on a T of "anythings": T<?>. Cannot now call T.method that takes a ? since we don't know what type is required, but can consume a ? (common superclass Object).
Can create T<?>[] somehow..
Have bounded types: <S extends T>, <S super T>
<S super T> S A<T>.doStuff() is a compile time error due to type inference limitations
Can use Class<T> newInstance() to get static typing, good for the exam!

Black Box

Try all possible inputs / outputs and validate they are correct
Can get full coverage
Impossible, esp. stateful progs.
Boundary value analysis checks inputs in pathological cases only

White Box

Examine structure of the code and try patterns so as to exercise every reasonable code path
Might get closer to full coverage than with black box testing
Takes advantage of the knowledge of internals to avoid pointlessly similar test cases
Number of unique paths is vast
A bug might be that a path is missing: check vs. the spec.

Code Inspection

Group exercise with programmer, spec-writer, test engineer, moderator
Check code against common error checklist (e.g. fencepost)
Teaches programmers to think critically and shares expertise
Walkthrough: testing done via a small number of test cases, participants trace the execution

Coverage

Statement: execute each statement just once, doesn't test control paths at all!
Decision: demonstrate true/false at each choice point (includes looping): what about multi-way branching or zero decisions?
Condition: at each branch each Boolean variable should take on both true and false at least once in the test cases: fails to explore some branches of the code
Decision-Condition: requires sufficient test cases to explore all branches and all assignments of Boolean variables in conditions: what about shortcut evaluation?
Multiple-Condition: requires test cases are not weakened by shortcut evaluation (= more test)

Equivalence Partitioning

Good test cases reduce by more than 1 the number of other cases that must be written and give information about a range of input: reduce inputs into equivalence classes that will find the same bugs
From the spec determine valid/invalid input equivalence classes then write tests to cover as many of the valid input equivalence classes as possible at once and tests to find exactly one of the invalid input equivalence classes

Other Tests

Facility, volume, stress, usability, security, performance, storage, configuration, compatibility, installability, reliability, recovery, serviceability, documentation, procedure, acceptance