5.1

A class is a concrete blueprint from which objects can be created.

An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without a body).

An interface defines what methods a class must implement, without specifying how.

5.2

Dynamic polymorphism:

Run the method in the child

Must be done at run-time since that's when we know the child's type

e.g.

```
    Animal a = new Dog(); //Animal reference, Dog object
    a.makeSound(); //Calls dog's makeSound(), not Animal's
```

Useful because additional subclasses can be added without modifying the existing code. Also, the code can be reused.

5.3

The whole point of inheritance is that a child class 'is a kind of' parent class, which means that all methods in the parent class are inherited by the child class. If the child class does not inherit every method of the parent class then the 'is a' relationship breaks.

5.4

Please find the file on git as well.

```
    public abstract class Student {
    protected int ticks;

    public Student(int ticks) {
        this.ticks = ticks;
    }
    public abstract boolean pass(); //this is now polymorphic
}
public class CSStudent extends Student {
    public CSStudent(int ticks) {
        super(ticks);
    }
    @Override
    public boolean pass() {
        return ticks >= 20;
    }
}
public class NSStudent extends Student {
    public NSStudent(int ticks) {
        super(ticks);
    }
    @Override
    public boolean pass() {
```

```
        return ticks >= 10;
    }
}
void main() {
    List<Student> students = new ArrayList<>();
    students.add(new CSStudent(18));
    students.add(new NSStudent(12));
    students.add(new CSStudent(21));

    for (Student student : students) {
        System.out.println(student.pass());
    }

}
```

5.5

The same way we created an abstract method pass() in the previous question, we can create an abstract method getType() in class Shape. Then all the subclasses can override it and output the corresponding to them type.

5.8

Please find the copy of the code on git

```
    public class Employee {
    void work(){
        System.out.print("Doing something employeelike.");
    }
public class Ninja {
    void sneak(){
        System.out.print("Being very sneaky...");
    }

}
public interface Ninjalike {
    void sneak();
}
public class NinjaEmployee extends Employee implements Ninjalike{
    private Ninja ninja = new Ninja();

    @Override
    public void sneak(){
        ninja.sneak();
    }
}
```

6.1

(a)

Mark: Walk the object graph and mark reachable objects. Sweep: Traverse all objects in memory, freeing unmarked ones and adding their regions to a free-list.

Mark cost is similar for all three approaches because all three must traverse reachable objects.

Sweep cost is potentially high, because the garbage collection must scan the entire heap to find unmarked objects, and each freed chunk must be recorded in the free-list. This leaves "holes" in memory. Allocation later may be slower because the system must search the free-list to find a suitably sized chunk.

(b)

Mark reachable objects. Sweep unmarked objects (as above). Move surviving objects to eliminate gaps.

Compacting requires moving every surviving object, which means copying data and updating all references which is expensive, particularly when many objects survive.

There is no free memory between the objects after compacting. Memory becomes one large contiguous block which means allocation becomes fast.

(c)

Can we go over this in SV, please?

6.2

Making classes immutable means that there are fewer short lived objects created which means the frequency of garbage collection is lower. So I would say that garbage collection benefits from immutable classes (as in there is not much work to do).

6.4

The marker interface does not declare any attributes or methods. Its only purpose is to label a class as having some sort of behaviour without implementing any specific methods. The marker interface tells how to handle the class.

6.5

Not calling super.clone() causes inherited fields to not be cloned properly. Moreover, manually creating a new object calls the constructor, which might reset or lose important state. Please see code example in the repository. In the example, both outputs are the same but if we had other private fields that are set dynamically during runtime, then those fields could be lost or incorrectly reinitialised because we call the constructor instead of super.clone().

6.6

(a)

See the code in repository

(b)

See the code in repository

(c)

Copy constructors do not work well with inheritance hierarchies. We have to explicitly call the superclass copy constructor manually, which is error prone.

(d)

When inheritance is not involved?...

6.7

The field is final which means that once it is assigned, it cannot be reassigned.

7.1

LinkedList: linked list of elements

Access by index: O(n)

Insertions/removals at beginning or end: O(1)

Insertions/removals in the middle: O(n) (need to traverse)

ArrayList: array of elements (efficient access)

Access by index: O(1)

Insertions/removals at the end: O(1)

Insertions/removals in the middle: O(n)

Vector: legacy class, as ArrayList but threadsafe.

Access by index: O(1)

Insertions/removals at the end: O(1)

Insertions/removals in the middle: O(n)

All of those allow duplicates.

TreeSet on the other hand does not allow duplicates because it enforces set semantics.

7.5

Each call to new String("Hi") creates a new object on the heap, even if the content is the same. So s1 refers to one object while s2 refers to a different object. Since == checks whether two references point to the same object, the result is false.

Literal "Hi" is created once, then reused. So s3 and s4 both point to the same String. Again, == compares references, and both references point to the same object.