Other stuff

Iterator – Can be made with .iterator() for collections, creates Iterator object which can be used to iterate through a collection, using .next() – returns an Object, or .hasNext() – checks if it has another object.
Primitive – Types that aren't considered to be objects and represent raw values
If -1 returned for compareTo: this < otherObject
For hashCode() make it based off fields not using Objects.hash(), like field1 + field2. Two objects can have the same hashCode and not be equal but two equal Objects must have the same hashCode.

Debugging

Defect: Error in code created by programmer
Infection: Error in program state
Propagation – Bad program state leads to more bad states
Failure – Program finally does something wrong

Polymorphism

Static type – The declared type, used by compiler to determine method calls
Dynamic type – what it's actually type is, determined when object is created, dynamic type is always a subtype of the static type. Determines what method is invoked at runtime

When choosing what method to execute when overloaded method is called chooses it based on the static type of the parameter being passed in.

Overloading:
When a method has the same name but different parameter types as another method in the same class (or possibly a superclass).

Can call methods from superclass using super.methodName(), super() – super constructor

Overriding doesn't work if return type is different and program won't compile.

Always call super() for a subclass constructor

Generics

T[] items = (T[]) new Object[10];

A wildcard type represents a concrete type which is unknown
Assuming you are getting Shape objects out of the list
Upper Bound: <? extends Shape>
The type must be a subtype of Shape (including Shape)
We can use any methods in Shape on the objects if we have a List, we can't add anything (since we don't know what it is)

Lower Bound: <? super Shape>
The type must be Shape or a superclass of Shape
We can add Shapes to a List<? super Shape> but we shouldn't do anything with the list, since it might be a List<Object>

Generic method
public <T> void toList(T[] a, Collection<T> c) {

…
}

Exceptions

Using return values which represent exceptions can be messy and often get confused with normal code, so its best to use exceptions.

When calling a method which throws an exception need to put method call in try {} and have a following catch(exceptionName e) {…} for catching the exception

finally {} - useful for cleaning up allocated resources that need to be cleaned up

Unchecked exceptions:
-Subclasses of RuntimeException **or** Error (depending on exception, assume exception extends RunTimeException)
-E.g. NullPointerException and IndexOutOfBoundsException both these extend RunTimeException not error

Checked exceptions:
-Subclasses of Exception but not RuntimeException
-E.g. IOException
-Must use "throws exceptionName" next to method name if method throws checked exception, the class calling that method must catch the exception or also have "throws exceptionName" (method only)

Key difference between checked and unchecked exceptions is checked exceptions signal recoverable problems (programs can typically recover from the error) and needed to be caught (try-catch) or thrown (throws) whereas unchecked exceptions signal observed bug (albeit not always fatal) and don't need to be caught/thrown.

class exceptionTest extends Exception/RunTimeException {…} for writing a new exception for constructors just call super(), super(String s), super(String s, Throwable t) depending on constructor

Throwable is checked exception

Testing

Black-box testing: Testing without knowledge of the implementation, test cases generated directly from specification, unbiased approach, robust to implementation changes

White-box testing: Testing with complete knowledge of implementation, test cases generated by looking at program code, aim to reach high-degree of code coverage, potentially biased approach, not robust to implementation changes

Bias
Programmer may misinterpret specification, may believe a particular part is well-coded so they don't write test cases for the part, programmer unlikely to represent target audience.

Function Coverage: number of methods invoked / # methods
Statement Coverage: number of statements executed / # statements
Branch Coverage: number of branches where both true and false cases are tested / # branches

Infeasible path is a path that is impossible to reach due to the logic of the function.

Simple execution path
Definition: a path through the method which iterates each loop at most once.
Needed as in some cases due to polymorphism/loops you could have an infinite amount of paths to test for (including future changes)

Inner classes/Anonymous classes/Inner static classes

Static inner classes (can be inside class/interface)
class Foo { public static class Bar {...} } – Declaring the static class Bar inside Foo
Foo.Bar bar = new Foo.Bar(); - creating object from static class
static class same for all objects, property of the class

Inner classes
Class can be private/public
e.g.

```
class IntList implements Iterable<Integer> {
   private int[] data;
   private class InternalIter implements Iterator<Integer>{
       IntList.this.data[InternalIter.this.pos++]; - explicitly gets data from IntList (its outer pointer), and
pos from InternalIter (itself)
   }
```

So to get field explicitly its className.this.fieldName

Static doesn't have outer pointer non-static does (so static can't access enclosing info), static object can be constructed without outer pointer, non-static can't.

Anonymous classes
Has no class definition and name
Can be used to instantiate interfaces assuming it implements required methods.
Defined as extension of existing class, can overwrite methods and/or define fields. **E.g.**

```
List myList = new ArrayList(){
//override ArrayList.add
   public boolean add(String x) {
       System.out.println("ADDED: " + x);
       return super.add(x);
   }
};
```

Serialization

Process of converting object into byte sequence – Can write sequence to file and reload it later!
Class/interface must implement/extend Serializable

Reflection

Definition – the ability of a program to examine and possibly modify its high level structure at runtime. Useful for working with anonymous objects (objects whose class is not known at runtime)

String.class – gets String class

String s1 = new String("X");
Class<? extends String> c1 = s1.getClass(); - gets class of s1
System.out.println(c1.getName()) – prints “String”
Method m[] = c1.getMethods(); - gets all methods of class
Reflection gives access to metadata **(data about data)**
Can use reflection to get fields (even private ones) (getDeclaredField(String s)), can set fields as well.

Object contracts

For determining if an object equals another (.equals())
reflexive
symmetric
transitive
consistent – for any non-null references x and y multiple invocations of x.equals(y) will consistently return the same result.
If x doesn’t equal null and y equals null x.equals(y) should always return false

Encapsulation

An Object’s implementation can change without affecting rest of system
Implementation must be invisible from outside
Each change should keep the object consistent
The objects invariant must be maintained
Way of encapsulating – don’t use public fields, use private fields and getters/setters methods instead

Cloning

Shallow clone – copies the data, if there are references to objects, copies of these references are created rather then creating cloned objects so changes in the original object are reflected in the cloned object and vice-versa.

Deep clone – copies the data, any references to objects are cloned as well (new cloned objects created) resulting in the cloned object being disjoint from the original object.

Java 8

Default methods interfaces
A default implementation for a method inside an interface, can be overridden by classes implementing this interface, just add default keyword before return type of method.

Static methods interfaces
Static method in an interface, good for declaring “pre-defined” implementations of the method, can’t be overridden

Lambdas
Pretty much a nested anonymous class but using different (condensed) syntax. Lambdas are for **functional interfaces**
Functional interfaces – an interface that has exactly one abstract (unimplemented) method.

```
Collections.sort(ls, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
```

```
    }
} );
```

To

```
Collections.sort(ls, (s1,s2)->s1.compareToIgnoreCase(s2));
```

Optional
Just a variable that may or may not be initialised, to determine if it is use isPresent(). Optional<T> is an alternative instead of using null.

Streams
.stream() – create stream of object
.parallelStream() – create parallel stream (uses multiple processors)
Intermediate operations
.filter(p -> p < 3) – filters data to only contain integers (assuming p is int) that are less than 3
.map(student -> student.age) – will transform Student objects into the age field of student
.sorted() – sort the stream in its natural order
Terminal operations
.forEach(p -> System.out.println(p)) – will perform an action for each element in the stream (simplified for loop), will print every p in this case.
.reduce(0, (a,b) -> a + b) – perform reduction on elements of the stream, reduces elements to 0
.collect(Collectors.toList()) – converts stream to a collection (in this case list), i.e. collects elements into a collection.

Garbage Collection

Reclamation of memory occupied by unreachable objects since these objects don't affect program execution. An object is unreachable if it isn't stored in a local/static variable, field and array element of a reachable object.

Pros: Don't have to explicitly free memory, performance is improved in the general case
Cons: Performance can be worse for simple programs, garbage collection isn't instant, system is paused during garbage collection and these pauses are unpredictable (issue for real time systems)

Mark 'n Sweep
Unreachable and reachable objects are mixed up together, Mark 'n Sweep separates these objects by sweeping reachable objects to the left and unreachable objects to the right.

System.gc() – force garbage collection (not guaranteed to work)