#Good to remember

//Equals method

Guidelines for Overriding equals:
It's crucial to adhere to these principles:

1. Reflexive: x.equals(x) should always be true.
2. Symmetric: If x.equals(y) is true, then y.equals(x) must also be true.
3. Transitive: If x.equals(y) and y.equals(z) are true, then x.equals(z) must be true.
4. Consistent: Multiple calls to x.equals(y) should consistently return the same result as long as neither object is modified.
5. Null Comparison: x.equals(null) should always return false.

When to use get.class or instanceof:
instanceof: Allows for polymorphic equality!
getClass() != obj.getClass(): Ensures strict type equality!

Let's consider a Person class:

```
class Person {
    private String name;
    private int age;

    // Constructor, getters, etc.
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Reflexive
        if (obj == null || !(obj instanceof Person)) return false; // Null check & type check
        Person other = (Person) obj;
        return age == other.age && Objects.equals(name, other.name); // Check field values
    }
    // It's highly recommended to override hashCode() when overriding equals()
}

//you can use super if super has also overwritten equals
if (!super.equals(obj)) return false;
if (!(obj instanceof Child)) return false;
Child other = (Child) obj;
return height == other.height;
```

//Custom Exceptions

Types:
Checked (Exception): Must be caught or declared, for recoverable errors.
Unchecked (RuntimeException): Don't require explicit handling, for programming errors.

Example:

```
// Checked/Unchecked
public class MyExcep/MyError extends Exception/RuntimeException {
   public MyExcep/MyError(String message, Throwable cause){
        super(message, cause);
    }
}
//Use in method: das in Klammern nur bei Checkt, dann natürlich OHNE die Klammern!
public void someMethod(int val) (throws MyExecp){
    if (val < 1 || val > 5) {
        throw new CustomException("Value must be between 1 and 5 (inclusive)");
    }
}
```

Usage:
You only need to do this with Checked Exceptions! You can with Unchecked but don't have to.

```
try {
    someMethod(3); // Could throw CustomCheckedException
} catch (CustomCheckedException e) {
    // Handle the exception here
}
```

//Iterator

The `Iterator` interface in Java defines three key methods:

1. `hasNext()`: Returns `true` if there are more elements to iterate over, and `false` if the iteration is complete.
2. `next()`: Returns the next element in the iteration sequence. If there are no more elements, it throws a `NoSuchElementException`.
3. `remove()`: (Optional) Removes the last element returned by `next()` from the underlying collection. Note that not all collections support this operation.

Examples:

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");

Iterator<String> iterator = names.iterator();

while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name); // Output: Alice, Bob, Charlie
}
```

Important Considerations:

- **Concurrent Modification:** Modifying the underlying collection while iterating using an iterator can lead to a `ConcurrentModificationException`. Use the iterator's `remove()` method or other safe modification techniques to avoid this.
- **Iterator Types:** Some collections offer specialized iterators like `ListIterator` (for lists), which allows bidirectional traversal and modification.
- **Java 8 Enhancements:** Java 8 introduced the `forEachRemaining()` method in the `Iterator` interface and functional iteration mechanisms like the enhanced `for` loop and lambda expressions, which provide more concise ways to work with iterators.

//Comparable and Comparator

Comparable Interface

- **Purpose:** Defines the *natural ordering* for a class.
- **Implementation:** A class implements `Comparable` and provides a `compareTo(T obj)` method. This method compares the current object (`this`) to another object of the same type (`obj`) and returns:
    - **Negative integer:** If the current object is less than `obj`.
    - **Zero:** If the current object is equal to `obj`.
    - **Positive integer:** If the current object is greater than `obj`.
- **Use Case:** You want to define a single, default way to sort objects of a class. This is useful when there's a clear, inherent order.

Comparator Interface

- **Purpose:** Provides a way to define *custom ordering* for objects.
- **Implementation:** You create a separate class that implements `Comparator`, providing a `compare(T o1, T o2)` method. This method compares two objects (`o1` and `o2`) and returns:
    - **Negative integer:** If `o1` is less than `o2`.
    - **Zero:** If `o1` is equal to `o2`.
    - **Positive integer:** If `o1` is greater than `o2`.
- **Use Case:** You want flexibility in how objects are sorted, perhaps needing to sort them based on different criteria at different times.

Key Differences:

Feature	Comparable	Comparator
Interface	`java.lang.Comparable<T>`	`java.util.Comparator<T>`
Implementation	Within the class itself	In a separate class
Ordering	Single, natural ordering	Multiple, custom orderings
Method	`compareTo(T obj)`	`compare(T o1, T o2)`
Use Case	Default sorting, one inherent order	Flexible sorting, multiple sorting criteria

Example:

```
class Person implements Comparable<Person> {
    private int age;
    private String name;

    @Override
    public int compareTo(Person other) {
        // Compare by age first (primary sorting criterion)
        int ageComparison = Integer.compare(this.age, other.age);
        if (ageComparison != 0) return ageComparison;
        // If ages are equal, compare by name (secondary sorting criterion)
        return this.name.compareTo(other.name);
    }
}

// Custom ordering (Comparator)
class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.name.compareTo(p2.name); // Sort by name
    }
}

public class SortingDemo {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        // Add Person objects to the list

        Collections.sort(people); // Uses Person's natural ordering (by age)

        NameComparator nameComparator = new NameComparator();
        Collections.sort(people, nameComparator); // Sort by name (using Comparator)
    }
}
```

//Enums

- **Methods:** Enums can have methods and constructors.
- **Fields:** Enums can have fields (variables).
- **Iteration:** You can easily iterate over all the values of an enum.
- **Built-in Methods:** Enums come with helpful methods like values() (returns an array of all enum constants) and valueOf() (returns an enum constant based on its name).

Example:

```
enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double getMass() {
        return mass;
    }

    public double getRadius() {
        return radius;
    }

    // Method to find planet by mass and radius
    public static Planet findByMassAndRadius(double mass, double radius) {
        for (Planet planet : Planet.values()) {
            if (Math.abs(planet.getMass() - mass) < 1e18 &&
                Math.abs(planet.getRadius() - radius) < 1e3) {
                return planet;
            }
        }
        return null; // Return null if no match is found
    }
}
```

//printf

```
System.out.printf(format, arg1, arg2, ...);
```

Specifier	Description	Example	Output
%d	Integer (decimal)	%d, 10	10
%f	Floating-point number	%.2f, 3.14159	3.14
%s	String	%s, "Hello"	Hello
%c	Character	%c, 'A'	A
%b	Boolean (true or false)	%b, true	true
%n	Line separator	%n	(newline)

//Collections

Interface/Class	Description	Ordering	Duplicate	Example Uses	Performance Notes
List	Ordered collection, allows duplicates	Maintains insertion order	Yes	Storing elements in a specific sequence, maintaining history	Fast access by index, slow insertion/deletion in the middle
ArrayList	Resizable array implementation of List	Maintains insertion order	Yes	General-purpose list, frequent access by index	Fast random access, slower resizing
LinkedList	Doubly-linked list implementation of List	Maintains insertion order	Yes	Frequent insertion/deletion at any position	Slower random access
Set	Unordered collection, no duplicates allowed	No inherent order	No	Storing unique elements, removing duplicates	Fast membership testing (contains)
HashSet	Hash table implementation of Set	No inherent order	No	General-purpose set, fast lookups	Relies on good hash function for performance
TreeSet	Sorted set implementation (using red-black tree)	Sorted according to natural ordering or comparator	No	Maintaining elements in sorted order	Slower than HashSet, but provides sorted order
Queue	Collection for holding elements in a FIFO manner	Implementation dependent	Yes	Processing tasks in order, breadth-first search	Efficient insertion/removal at both ends
PriorityQueue	Queue that orders elements based on priority	Determined by priority or comparator	Yes	Scheduling tasks, Dijkstra's algorithm	Efficient access to the highest/lowest priority element
Map	Stores key-value pairs, each key is unique	Implementation dependent	No	Associating values with keys, dictionaries, caches	Fast lookups by key
HashMap	Hash table implementation of Map	No inherent order	No	General-purpose map, fast lookups	Relies on good hash function for performance
TreeMap	Sorted map implementation (using red-black tree)	Sorted by keys' natural ordering or comparator	No	Maintaining key-value pairs in sorted order	Slower than HashMap, but provides sorted order
Deque	Double-ended queue, access/insert at both ends	Maintains insertion order	Yes	Stack, queue implementations	Efficient insertion/removal at both ends