Cheat Sheet for comprehensive Oracle Certified Professional- Java SE 11 Developer

Java Basics

Data Types

- **Primitive Types**: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
- **Reference Types**: `String`, `Object`, `Arrays`, `Enums`

Variables

- **Declaration**: `int age;`
- **Initialization**: `age = 30;`
- **Combined**: `int age = 30;`

Operators

- **Arithmetic**: `+`, `-`, `*`, `/`, `%`
- **Relational**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logical**: `&&`, `||`, `!`
- **Assignment**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

Control Flow

Conditional Statements

- **if-else**:

```
if (condition) {
    // code
} else if (anotherCondition) {
    // code
} else {
    // code
}
```

- **switch**:

```
switch (expression) {
    case value1:
        // code
```

```
            break;
        case value2:
            // code
            break;
        default:
            // code
    }
```

Loops

- **for**:

```
    for (int i = 0; i < 10; i++) {
        // code
    }
```

- **while**:

```
    while (condition) {
        // code
    }
```

- **do-while**:

```
    do {
        // code
    } while (condition);
```

Object-Oriented Programming (OOP)

Classes and Objects

- **Class Declaration**:

```
    public class MyClass {
        // fields, constructors, methods
    }
```

- **Object Creation**:

```
    MyClass obj = new MyClass();
```

Constructors

- **Default Constructor**:

```
public MyClass() {
    // code
}
```

- **Parameterized Constructor**:

```
public MyClass(int value) {
    // code
}
```

Methods

- **Declaration**:

```
public void myMethod() {
    // code
}
```

- **Method Overloading**:

```
public void myMethod(int value) {
    // code
}
```

Inheritance

- **Extends**:

```
public class SubClass extends SuperClass {
    // code
}
```

- **Overriding**:

```
@Override
public void myMethod() {
    // code
}
```

Polymorphism

- **Method Overriding**:

```
public class SubClass extends SuperClass {
    @Override
    public void myMethod() {
        // code
    }
}
```

- **Method Overloading**:

```
public void myMethod(int value) {
    // code
}
```

Encapsulation

- **Access Modifiers**: `public`, `private`, `protected`, `default`

- **Getters and Setters**:

```
public int getValue() {
    return value;
}
public void setValue(int value) {
    this.value = value;
}
```

Abstraction

- **Abstract Class**:

```
public abstract class MyAbstractClass {
    public abstract void myMethod();
}
```

- **Interface**:

```
public interface MyInterface {
    void myMethod();
}
```

Exception Handling

Try-Catch-Finally

- **Basic Structure**:

```
try {
    // code
} catch (ExceptionType e) {
    // code
} finally {
    // code
}
```

Throw and Throws

- **Throw**:

```
throw new ExceptionType("Message");
```

- **Throws**:

```
public void myMethod() throws ExceptionType {
    // code
}
```

Collections Framework

Collection Interfaces

- **List**: `ArrayList`, `LinkedList`
- **Set**: `HashSet`, `TreeSet`
- **Map**: `HashMap`, `TreeMap`

Common Methods

- **Add**: `add()`, `put()`
- **Remove**: `remove()`, `clear()`
- **Iterate**: `for-each`, `Iterator`

Generics

Generic Classes

- **Declaration**:

```
public class MyClass<T> {
    private T value;
    public T getValue() {
        return value;
    }
    public void setValue(T value) {
        this.value = value;
    }
}
```

Generic Methods

- **Declaration**:

```
public <T> void myMethod(T value) {
    // code
}
```

Lambda Expressions

Basic Syntax

- **Lambda**:

```
(parameters) -> expression
(parameters) -> { statements; }
```

Functional Interfaces

- **Predicate**: `Predicate<T>`
- **Function**: `Function<T, R>`
- **Consumer**: `Consumer<T>`
- **Supplier**: `Supplier<T>`

Stream API

Creating Streams

- **From Collection**:

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
```

- **From Array**:

```
Stream<String> stream = Stream.of("a", "b", "c");
```

Common Operations

- **Filter**: `filter(Predicate<T>)`
- **Map**: `map(Function<T, R>)`
- **Reduce**: `reduce(BinaryOperator<T>)`
- **Collect**: `collect(Collector<T, A, R>)`

Concurrency

Threads

- **Creating Threads**:

```
Thread thread = new Thread(() -> {
    // code
});
thread.start();
```

Executors

- **ExecutorService**:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() -> {
    // code
});
```

Synchronization

- **Synchronized Methods**:

```
public synchronized void myMethod() {
    // code
}
```

- **Synchronized Blocks**:

```
synchronized (lock) {
    // code
}
```

I/O and NIO

File I/O

- **Reading**:

```
try (BufferedReader reader = new BufferedReader(new
FileReader("file.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        // code
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

- **Writing**:

```
try (BufferedWriter writer = new BufferedWriter(new
FileWriter("file.txt"))) {
    writer.write("Hello, World!");
} catch (IOException e) {
    e.printStackTrace();
}
```

NIO (New I/O)

- **Path**:

```
Path path = Paths.get("file.txt");
```

- **Files**:

```
List<String> lines = Files.readAllLines(path);
```

Modules

Module Declaration

- **module-info.java**:

```
module my.module {
    requires module.name;
    exports package.name;
}
```

Module Path

- **Using Modules**:

```
java --module-path mods --module my.module/com.example.Main
```

Miscellaneous

Annotations

- **Custom Annotation**:

```
@interface MyAnnotation {
    String value();
}
```

- **Usage**:

```
@MyAnnotation("value")
public void myMethod() {
    // code
}
```

Date and Time API

- **LocalDate**:

```
LocalDate date = LocalDate.now();
```

- **LocalTime**:

```
LocalTime time = LocalTime.now();
```

- **LocalDateTime**:

```
LocalDateTime dateTime = LocalDateTime.now();
```

Optional

- **Creating Optional**:

```
Optional<String> optional = Optional.of("value");
```

- **Checking Value**:

```
if (optional.isPresent()) {
    // code
}
```

- **Default Value**:

```
String result = optional.orElse("default");
```

Tips and Tricks

- **Use `final` for constants**: `public static final int MAX_VALUE = 100;`
- **Avoid `null`**: Use `Optional` instead.
- **Use `StringBuilder` for string concatenation in loops**.
- **Leverage Stream API for complex data processing**.
- **Use `try-with-resources` for automatic resource management**.
- **Follow naming conventions**: Classes start with uppercase, methods with lowercase.
- **Use `@Override` annotation for overridden methods**.
- **Use `var` for local variable type inference**: `var list = new ArrayList<String>();`

Summary

- **Java Basics**: Data types, variables, operators, control flow.
- **OOP**: Classes, objects, inheritance, polymorphism, encapsulation, abstraction.

- **Exception Handling**: Try-catch-finally, throw, throws.
- **Collections**: List, Set, Map, common methods.
- **Generics**: Generic classes, methods.
- **Lambda Expressions**: Basic syntax, functional interfaces.
- **Stream API**: Creating streams, common operations.
- **Concurrency**: Threads, executors, synchronization.
- **I/O and NIO**: File I/O, NIO basics.
- **Modules**: Module declaration, module path.
- **Miscellaneous**: Annotations, Date and Time API, Optional.
- **Tips and Tricks**: Best practices, naming conventions, resource management.

By Ahmed Baheeg Khorshid

ver 1.0