Cheat Sheet for comprehensive Oracle Certified Professional- Java SE 8 Programmer

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 (variable) {
    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**:

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

- **Object Creation**:

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

Constructors

- **Default Constructor**:

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

- **Parameterized Constructor**:

```
MyClass(int param) {
    // code
}
```

Methods

- **Declaration**:

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

- **Overloading**:

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

Inheritance

- **Extends**:

```
class ChildClass extends ParentClass {
    // code
}
```

- **Overriding**:

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

Polymorphism

- **Method Overriding**:

```
class Parent {
    void show() {
        // code
    }
}
class Child extends Parent {
    @Override
    void show() {
        // code
    }
}
```

Abstraction

- **Abstract Class**:

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

- **Interface**:

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

Exception Handling

try-catch-finally

- **Basic Structure**:

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

Throwing Exceptions

- **throw**:

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

Custom Exceptions

- **Declaration**:

```
class MyException extends Exception {
    MyException(String message) {
        super(message);
    }
}
```

Collections Framework

List

- **ArrayList**:

```
List<String> list = new ArrayList<>();
list.add("item");
```

- **LinkedList**:

```
List<String> list = new LinkedList<>();
list.add("item");
```

Set

- **HashSet**:

```
Set<String> set = new HashSet<>();
set.add("item");
```

- **TreeSet**:

```
Set<String> set = new TreeSet<>();
set.add("item");
```

Map

- **HashMap**:

```
Map<String, Integer> map = new HashMap<>();
map.put("key", 1);
```

- **TreeMap**:

```
Map<String, Integer> map = new TreeMap<>();
map.put("key", 1);
```

Generics

Generic Classes

- **Declaration**:

```
class MyGenericClass<T> {
    T field;
    void setField(T value) {
        this.field = value;
    }
}
```

Generic Methods

- **Declaration**:

```
public <T> void myGenericMethod(T param) {
    // code
}
```

Lambda Expressions

Basic Syntax

- **Lambda**:

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

Functional Interfaces

- **Predicate**:

```
Predicate<String> predicate = s -> s.length() > 5;
```

- **Function**:

```
Function<String, Integer> function = s -> s.length();
```

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");
```

Intermediate Operations

- **filter**:

```
stream.filter(s -> s.startsWith("a"));
```

- **map**:

```
stream.map(String::toUpperCase);
```

Terminal Operations

- **forEach**:

```
stream.forEach(System.out::println);
```

- **collect**:

```
List<String> result = stream.collect(Collectors.toList());
```

Concurrency

Threads

- **Creating Threads**:

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

ExecutorService

- **Basic Usage**:

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

Synchronization

- **Synchronized Methods**:

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

- **Synchronized Blocks**:

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

I/O and NIO

File I/O

- **Reading File**:

```
  try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
      String line;
      while ((line = br.readLine()) != null) {
          System.out.println(line);
```

```
    }
}
```

- **Writing File**:

```
try (BufferedWriter bw = new BufferedWriter(new
FileWriter("file.txt"))) {
    bw.write("Hello, World!");
}
```

NIO (New I/O)

- **Path**:

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

- **Files**:

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

Annotations

Built-in Annotations

- **@Override**:

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

- **@Deprecated**:

```
@Deprecated
void oldMethod() {
    // code
}
```

Custom Annotations

- **Declaration**:

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

- **Usage**:

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

Miscellaneous

Date and Time API

- **LocalDate**:

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

- **LocalTime**:

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

- **LocalDateTime**:

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

Regular Expressions

- **Pattern and Matcher**:

```
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher("123");
boolean matches = matcher.matches();
```

Assertions

- **Basic Usage**:

```
assert condition : "Message";
```

Tips and Tricks

- **Use `final` for constants**:

```
final int MAX_VALUE = 100;
```

- **Avoid `null`**: Use `Optional` instead:

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

- **Use `StringBuilder` for string concatenation in loops**:

```
StringBuilder sb = new StringBuilder();
for (String s : list) {
    sb.append(s);
}
```

- **Use `try-with-resources` for automatic resource management**:

```
try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
    // code
}
```

Common Pitfalls

- **Null Pointer Exception**: Always check for `null` before accessing objects.
- **Infinite Loops**: Ensure loop conditions are correctly defined.
- **Resource Leaks**: Use `try-with-resources` for I/O operations.
- **Concurrent Modification**: Avoid modifying collections while iterating.

Exam Preparation

- **Practice**: Solve as many practice questions as possible.
- **Review**: Regularly review topics you find challenging.
- **Simulate**: Take mock exams to simulate the real exam environment.
- **Time Management**: Practice managing time effectively during exams.

This cheat sheet covers the essential topics and features required for the Oracle Certified Professional Java SE 8 Programmer exam. Use it as a quick reference to refresh your knowledge and prepare effectively.

By Ahmed Baheeg Khorshid

ver 1.0