Here are detailed and structured notes from the uploaded document:

Generics in Java

Introduction

- Introduced in **JDK 5**, Generics support abstraction over types (parameterized types) in classes and methods.
- **Purpose**:
  - Allows classes and methods to operate on any data type, specified during object instantiation or method invocation.
  - Improves type safety and reduces runtime errors.
- Example:

```
ArrayList<String> list = new ArrayList<>();
```

Features of Generics

1. **Type Safety**:
   - Enforces type restrictions, avoiding runtime errors like `ClassCastException`.
   - Example:

```
ArrayList<String> list = new ArrayList<>();
list.add(10); // Compile-time error.
```

2. **Simplified Code**:
   - Eliminates the need for explicit typecasting.
   - Example:

```
String value = list.get(0); // No typecasting required.
```

3. **Reusability**:
   - Generic code works for multiple data types.
4. **Error Detection at Compile-Time**:
   - Errors are caught during compilation rather than at runtime.

Syntax for Generic Classes

- **Single Parameter**:

```
class MyClass<T> {
    T obj;
}
MyClass<Integer> obj = new MyClass<>();
```

- **Multiple Parameters**:

```
class MyClass<T, V> {
    T obj1;
    V obj2;
}
MyClass<String, Integer> obj = new MyClass<>();
```

- **Diamond Syntax**:
    - Java infers the type from the declaration.

```
MyClass<String> obj = new MyClass<>();
```

Generic Methods

- Methods that work with any data type.
- Syntax:

```
public static <T> void display(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

- Invocation:

```
display(new Integer[]{1, 2, 3});
display(new String[]{"A", "B"});
```

Generic Constructor

- Allows parameterized types in a constructor.
- Example:

```
class MyClass {
    <T> MyClass(T obj) {
        System.out.println(obj);
    }
}
MyClass obj = new MyClass("Hello");
```

Generic Interfaces

- Defined using a type parameter, similar to generic classes.
- Syntax:

```
interface MyInterface<T> {
    void method(T t);
}
class MyClass implements MyInterface<String> {
    public void method(String t) {
        System.out.println(t);
```

```
    }
}
```

Bounded Type Parameters

- Restrict the data types that can be used.
- **Syntax**:

```
class MyClass<T extends Number> { }
```

- Multiple Bounds:

```
class MyClass<T extends Number & Comparable<T>> { }
```

Wildcard Characters (`?`)

- Used for unknown types in generics.

1. **Unbounded Wildcard (`<?>`)**:
    - Accepts any type.

```
void method(List<?> list) { }
```

2. **Upper Bounded Wildcard (`<? extends T>`)**:
    - Restricts to `T` and its subclasses.

```
void method(List<? extends Number> list) { }
```

3. **Lower Bounded Wildcard (`<? super T>`)**:
    - Restricts to `T` and its superclasses.

```
void method(List<? super Integer> list) { }
```

Advantages of Generics

1. **Reusability**:
    - Code works for multiple types.
2. **Type Safety**:
    - Prevents invalid data types.
3. **No Typecasting**:
    - Eliminates explicit type conversion.

Restrictions of Generics

1. Cannot use primitive types directly:
    - Use wrapper classes instead (e.g., `Integer` for `int`).
2. Cannot create objects of generic type:

```
T obj = new T(); // Invalid.
```

3. Static members cannot use generic types.
4. Cannot create arrays of generic types.

Key Points

1. Generics apply only at compile time.
2. At runtime, type information is erased (type erasure).
3. Bounded types use `extends` for both classes and interfaces.
4. Multiple bounds can include one class and multiple interfaces.

Example Programs

1. **Basic Generic Class**:

```
class MyClass<T> {
    T obj;
    MyClass(T obj) {
        this.obj = obj;
    }
    T getObj() {
        return obj;
    }
}
MyClass<Integer> obj = new MyClass<>(10);
System.out.println(obj.getObj());
```

2. **Generic Method with Wildcard**:

```
void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

3. **Generic Method with Bounded Wildcard**:

```
void addNumbers(List<? super Integer> list) {
    list.add(10);
}
```

Conclusion

- Generics improve code clarity, safety, and reusability.
- They are essential for modern Java programming, especially for collections like `ArrayList`, `HashMap`, etc.
- Runtime behavior does not differentiate between types due to type erasure.

These notes provide a comprehensive summary of Generics in Java. Let me know if you'd like further clarification or additional examples!