Generics in Java

Generics in Java

What are Generics?

Generics allow us to create classes, interfaces, and methods with a placeholder for types (like T, E, etc.).

Why use Generics?

Before Java 5, collections accepted any object-leading to type safety issues and runtime ClassCastException.

Example without Generics:

```
ArrayList list = new ArrayList();
list.add("Hello");
list.add(10); // no compile-time error

String str = (String) list.get(1); // Runtime error
```

Example with Generics:

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
String str = list.get(0); // No casting needed
```

Benefits:

- Type safety
- Compile-time checks
- No need for manual casting

Wrapper Classes in Java

Wrapper Classes in Java

What are Wrapper Classes?

They wrap primitive data types into objects. Needed because Java collections work only with objects.

Mapping:

int -> Integer

char -> Character

boolean -> Boolean

... and so on

Example:

```
int a = 5;
Integer obj = a; // Auto-boxing
int b = obj;     // Auto-unboxing
```

Benefits:

- Works with collections
- Has utility methods like Integer.parseInt()
- Can be null (unlike primitives)

Exception Handling in Java

Exception Handling in Java

What is Exception Handling?

A mechanism to handle runtime errors and maintain normal program flow.

Syntax:

try {

// code that might throw exception

} catch (ExceptionType e) {

// handling code

} finally {

// always executed

}

throw vs throws:

throw - used to throw an exception manually

throws - used in method signature to declare exception

Example:

throw new ArithmeticException("Divide by zero!");

public void readFile() throws IOException

Keywords:

try, catch, finally, throw, throws

Benefits:

- Prevents crashing

- Gives meaningful error messages

Lambda Functions in Java

Lambda Functions in Java

What is a Lambda?

A lambda expression is an anonymous function used to implement methods of functional interfaces.

Syntax:

(parameters) -> { body }

Example:

Runnable r = () -> System.out.println("Hello");

r.run();

Used with functional interfaces and streams.

Example with Stream:

List<Integer> nums = Arrays.asList(1, 2, 3);

nums.stream().filter(n -> n%2 == 0).forEach(System.out::println);

Benefits:

- Less boilerplate

- More readable

- Enables functional programming

Cloning of Objects in Java

Cloning of Objects in Java

What is Cloning?

Creating an exact copy of an object using the clone() method.

Steps:

1. Implement Cloneable interface

2. Override clone() and call super.clone()

Shallow vs Deep Cloning:

Shallow - copies references (not inner objects)

Deep - copies everything including nested objects

Shallow Clone:

```
class A implements Cloneable {
public Object clone() { return super.clone(); }
}
```

Deep Clone:

```
class Employee implements Cloneable {
Address addr;
public Object clone() {
Employee copy = (Employee) super.clone();
copy.addr = (Address) addr.clone();
return copy;
}
}
```

Benefits:

- Useful for making duplicates

- Saves object creation time