Generics and Exceptions

Generics

Generics were added to Java to provide compile-time type checking and removing the risk of ClassCastException that was common while working with collection classes. Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Generics allows us to reuse the same code with different inputs. We can use Java generics methods and classes for any type of objects. The idea is to allow type (Integer, String, ... etc, and user-defined types) to be a parameter to methods, classes, and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

Generic methods

We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

1. All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type.
2. Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

Let us say we want to write a generic method that can print an integer, string or any other type. For this, we will use the following syntax:

```
public static <T> void print(T t) {
    System.out.println(t);
}
```

Here, we have used the generic method print that can print any type of data. The type parameter is specified with the help of angle brackets. The type parameter section, delimited by angle brackets (< and >), follows the method name. It specifies the type parameters (also called type variables) T.

We can also use multiple type parameters in a generic method. For example:

```java
public static <T, U> void print(T t, U u) {
    System.out.println(t + " " + u);
}
```

Generic classes

If multiple methods are using generics then we can create a generic class that can be used by all those methods. For example, we can create a generic class Pair that can store a key-value pair. We can use this class to store a pair of integers, a pair of strings, etc.

```java
public class Pair<L, R> {
    private L left;
    private R right;

    public Pair(L left, R right) {
        this.left = left;
        this.right = right;
    }

    public L getLeft() { return left; }
    public R getRight() { return right; }
}
```

Apart from reducing type declarations, generics also allow us to create generic fields and constructors.

Bounded Type Parameters

We can also restrict the types that can be passed to a type parameter. For example, a method that works on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

If we want to write a method that can print the details of any type of employee, we can use the following syntax:

```java
public static <T extends Employee> void print(T t) {
    System.out.println(t);
}
```

We can also use multiple bounds on a type parameter. For example:

```
public static <T extends Employee & Comparable<T>> void print(T t) {
    System.out.println(t);
}
```

Wildcards

The question mark (?) represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

For example, we can write a method that works on lists of any type:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. In Java, an exception is an object that wraps an error event that occurred within a method and contains:

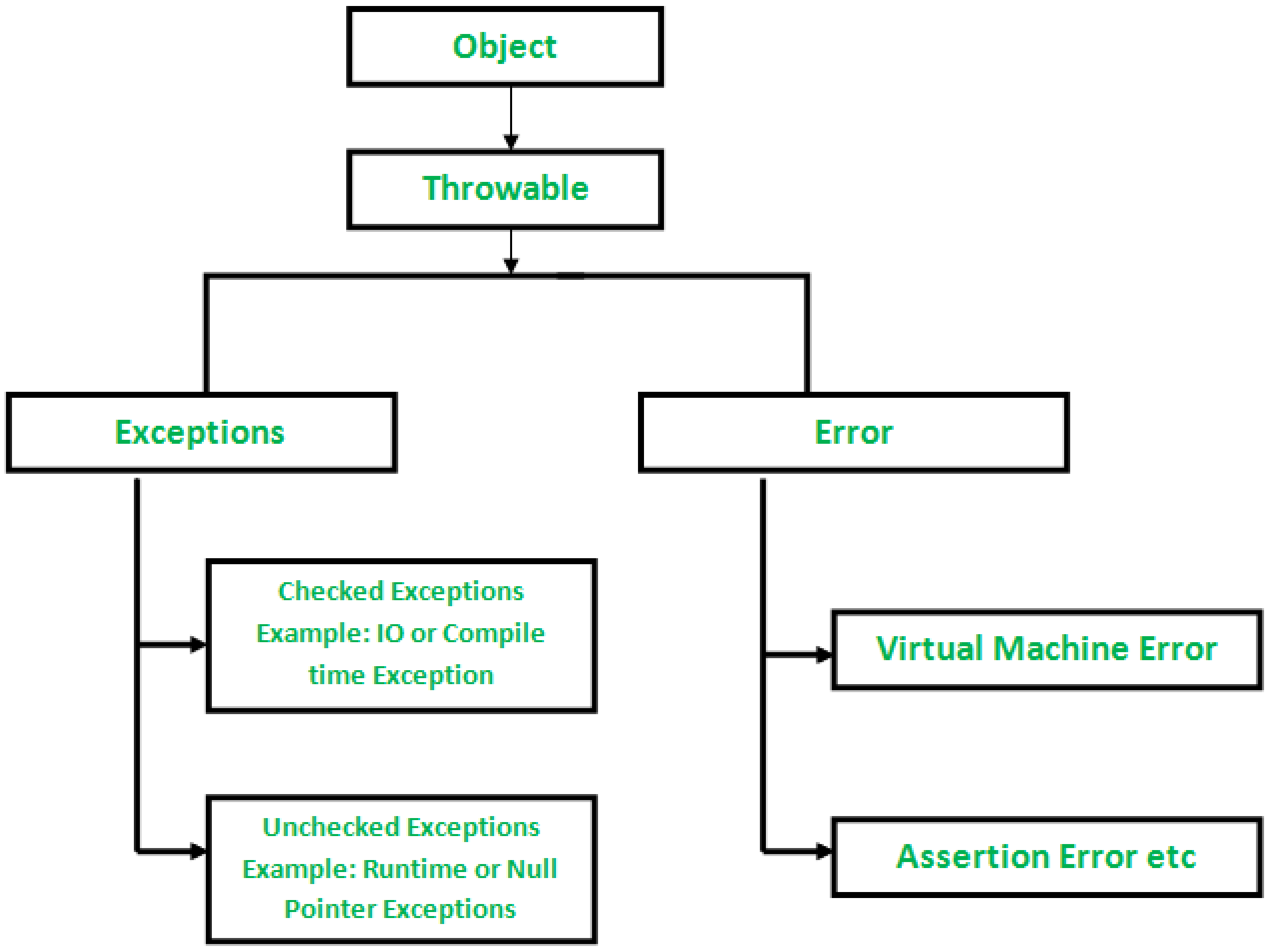
- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information about the error

Types of Exceptions

There are two types of exceptions:

- Checked exceptions are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. Example: IOException, SQLException, etc. Checked exceptions are also called as compile time exceptions.
- Unchecked exceptions are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers either handle them or let them propagate up. Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are also called as runtime exceptions.

Exception Hierarchy


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class, there is another subclass called Error which is derived from the Throwable class. Errors are not normally trapped by the Java programs. These conditions normally happen in case of severe failures, which are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally programs cannot recover from errors.

Exception Handling

The throw keyword

The throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

```
public int getPlayerScore(String playerFile) {
    if (playerFile == null) {
        throw new IllegalArgumentException("Player file cannot be null");
    }
    Scanner contents = new Scanner(new File(playerFile));
    return Integer.parseInt(contents.nextLine());
}
```

Since IllegalArgumentException is an unchecked exception, we don't need to declare it in the method signature. If we want to throw a checked exception, we need to declare it in the method signature.

The throws keyword

The simplest way to "handle" an exception is to rethrow it. This is done using the throws statement. The general form of throw is shown here:

```
public int getPlayerScore(String playerFile) throws FileNotFoundException
{

    Scanner contents = new Scanner(new File(playerFile));
    return Integer.parseInt(contents.nextLine());
}
```

The throws clause appears at the end of a method's signature. It informs the caller that the method might throw an exception. If it does, then the caller must either catch the exception or rethrow it. In the preceding example, the throws clause indicates that getPlayerScore( ) might throw a FileNotFoundException. Thus, any code that calls getPlayerScore( ) must either catch the exception or declare that it also can throw it.

The try and catch keywords

The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block. The try and catch keywords come in pairs:

```
public int getPlayerScore(String playerFile) {
    try {
        Scanner contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } catch ( FileNotFoundException noFile ) {
        logger.warn("File not found, resetting score.");
        return 0;
    }
}
```

The finally keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
public int getPlayerScore(String playerFile)
  throws FileNotFoundException {
    Scanner contents = null;
    try {
        contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } finally {
        if (contents != null) {
            contents.close();
        }
    }
}
```

The golden Rules of Exception Handling

1. Never swallow an exception - Swallowing an exception means that you catch it and do nothing with it. This is a bad practice because it means that you are ignoring the fact that an error occurred, which could cause your program to behave in unexpected ways and hide bugs.

```
try {
    // do something
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```

2. Never catch a generic exception - Catching a generic exception means that you are catching all exceptions, including runtime exceptions. This is a bad practice because it means that you are not handling exceptions in a meaningful way. You should always catch specific exceptions and handle them appropriately.

```
try {
    // do something
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```

3. Never throw a generic exception - Throwing a generic exception means that you are throwing all exceptions, including runtime exceptions. This is a bad practice because it means that you are not handling exceptions in a meaningful way. You should always throw specific exceptions and handle them appropriately.

```
public void doSomething() throws Exception {
    // do something
}
```