**Program errors and exception handling***•* Types of program errors  
*•* Syntax, semantic, and logical errors  
*•* Compile time and runtime errors  
*•* Test drivers  
*•* Debugging techniques  
*•* Exception handling

**1 Types of program errors**

We distinguish between the following types of errors:  
1. **Syntax errors**: errors due to the fact that the syntax of the language is not respected.  
2. **Semantic errors**: errors due to an improper use of program statements.  
3. **Logical errors**: errors due to the fact that the specification is not respected.  
From the point of view of when errors are detected, we distinguish:  
1. **Compile time errors**: syntax errors and static semantic errors indicated by the compiler.  
2. **Runtime errors**: dynamic semantic errors, and logical errors, that cannot be detected by the compiler (debugging).

**2 Syntax errors**Syntax errors are due to the fact that the syntax of the Java language is not respected.  
Let us see some examples of syntax errors.  
*Example 1:* Missing semicolon:  
int a = 5 // semicolon is missing  
Compiler message:  
Example.java:20: ’;’ expected  
int a = 5  
*Example 2:* Errors in expressions:

| x = ( 3 + 5;  y = 3 + \* 5; | // missing closing parenthesis ’)’ // missing argument between ’+’ and ’\*’ |
| --- | --- |

1  
2

**3 Semantic errors**Semantic errors indicate an improper use of Java statements.  
Let us see some examples of semantic errors.  
*Example 1:* Use of a non-initialized variable:  
int i;  
i++; // the variable i is not initialized  
*Example 2:* Type incompatibility:  
int a = "hello"; // the types String and int are not compatible  
*Example 3:* Errors in expressions:  
String s = "...";  
int a = 5 - s; // the - operator does not support arguments of type String  
*Example 4:* Unknown references:

| Strin x; | // Strin is not defined |
| --- | --- |
| system.out.println("hello"); // system is not defined |  |
| String s; s.println(); | // println is not a method of the class String |

*Example 5:* Array index out of range (dynamic semantic error)  
int[] v = new int[10];  
v[10] = 100; // 10 is not a legal index for an array of 10 elements  
The array v has been created with 10 elements (with indexes ranging from 0 to 9), and we are trying to access the element with index 10, which does not exist. This type of error is not caught during compilation, but causes an exception to be thrown at runtime.

**4 Logical errors**Logical errors are caused by the fact that the software specification is not respected. The program is compiled and executed without errors, but does not generate the requested result.  
Let us see some examples of logical errors:  
*Example 1:* Errors in the performed computation:  
public static int sum(int a, int b) {  
return a - b ;  
}  
// this method returns the wrong value wrt the specification that requires  
// to sum two integers  
*Example 2:* Non termination:  
String s = br.readLine();  
while (s != null) {  
System.out.println(s);  
} // this loop does not terminate

**5 Errors detected by the compiler and runtime errors**All syntax errors and some of the semantic errors (the static semantic errors) are detected by the compiler, which generates a message indicating the type of error and the position in the Java source file where the error occurred (notice that the actual error could have occurred before the position signaled by the compiler).  
Other semantic errors (the dynamic semantic errors) and the logical errors cannot be detected by the compiler, and hence they are detected only when the program is executed.  
Let us see some examples of errors detected at runtime:

*Example 1:* Division by zero:  
*Program errors and exception handling* 3  
int a, b, x;  
a = 10;  
b = Integer.parseInt(kb.readLine());  
x = a / b; //ERROR if b = 0  
This error occurs only for a certain configuration of the input (b = 0).

*Example 2:* File does not exist:  
FileReader f = new FileReader("pippo.txt");  
The error occurs only if the file pippo.txt does not exist on the harddisk.

*Example 3:* Dereferencing of a null reference:  
String s, t;  
s = null;  
t = s.concat("a");  
The concat() method cannot be applied to a reference whose value is null. Note that the above code is syntactically correct, since the concat() method is correctly applied to a reference of type String, but it contains a dynamic semantic error due the fact that a method cannot be applied to a reference whose value is null.

**6 Drivers for testing**Drivers for testing are program portions that are used to test the correctness of a class or of a method. The purpose of such drivers is to call all methods of the public interface of a class and verify that they respect the specification.  
In order to perform a test that is complete, we should follow some guidelines:  
*•* verify each functionality (each method);  
*•* perform the tests according to a specific order (the order of method application is often important);  
*•* ensure that each statement is executed at least once (for example, when we have a conditional statement, we have to perform the test for various configurations of the input, in such a way that the boolean condition becomes respectively true and false);  
*•* detect and test special cases (for example, an empty file as input to a method that reads from a file).

**7 Techniques for detecting errors (debugging)**

If the testing phase signals the presence of logical errors, or if we are not able to detect the cause for a runtime error, it is necessary to **debug** the program.  
There are two ways in which we can obtain information that is helpful for debugging a program:  
1. by inserting output statements in the code;  
2. by executing the program by means of a **debugger**.

**8 Debugging by inserting output statements**

This debugging technique is based on inserting in suitable positions of the source code statements that print the content of variables that could contain wrong values causing an error.  
*Example:*int a, b, x;  
a = 5;  
b = Integer.parseInt(kb.readLine()); // reading of b

| ...  x = a/b; | // statements that do not change b |
| --- | --- |

To *debug* this program statement we can verify, by printing the value of b on the screen, that the error occurs  
when the variable b has value 0.  
int a, b, x;  
a = 5;  
b = Integer.parseInt(kb.readLine()); // reading of b  
... // statements that do not change b  
System.out.println("b = " + b);  
x = a/b;

Once the causes for the error have been identified and corrected, the print statements can be removed before closing the debugging session.  
*Note:* If it is necessary to explore the content of objects, rather than of simple variables of primitive data types, we can make use of the toString() method, which provides information on the content of the object. We could also redefine toString() so as to simplify the reading of the state of the object during debugging.

**9 Execution of the program by means of a debugger**A debugger allows us to:  
*•* execute the statements of a program one at a time,  
*•* execute the program until the execution reaches certain points defined by the user (called **breakpoints**),  
*•* examine the content of variables and objects at any time during the execution.  
Debuggers are very useful tools to detect the causes of errors in programs.  
*Example 1:*int a, b, x;  
a = 5;  
b = Integer.parseInt(kb.readLine()); // reading of b

| ...  x = a/b; | // statements that do not change b |
| --- | --- |

By means of a debugger we can verify the value of the variable b before executing the statement that generates  
the error.  
*Example 2:*String s, t;  
s = null;  
...  
t = s.concat("a");  
The assignment statement for t generates an exception of type NullPointerException. Such an error depends on the fact that, when the assignment statement is executed, the value of s is null. To check this error, we can use a debugger and observe the value of the variable s before executing the statement that generates the error.

**10 Handling errors during the execution of programs**During the execution of a program, various conditions can occur that cause an unexpected and abnormal termination of the program.  
*Example:* Consider the following program:  
public class TestException {  
public static void main (String[] args) {  
int falseNumber = Integer.parseInt("OK");  
System.out.println("this println statement is not executed");  
}  
}  
the following message is printed on the screen:  
Exception in thread "main" java.lang.NumberFormatException: For input string: "OK"  
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)  
at java.lang.Integer.parseInt(Integer.java:468)  
at java.lang.Integer.parseInt(Integer.java:518)  
at TestException.main(TestException.java:3)

Hence, the string "this println statement is not executed" is not printed.  
In Java, the errors that occur at runtime are represented by means of **exceptions**. Java offers a predefined set of exceptions that can be thrown during program execution.  
To avoid that a program terminates unexpectedly, Java allows us to handle exceptions by means of a suitable construct.

**11 How to handle exceptions**

There are two possibilities for handling exceptions:

• handling the exception where it is generated

• handling the exception in another point of the program

If the exception is not handled, instead, the program terminates and prints out a message.

For example, if we try to do a division by zero, we obtain:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at DivByZero.main(DivByZero.java:7)

**12 The most common types of exceptions**

1. NullPointerException: generated when the reference used to invoke a method has value null, or when we try to access an instance variable through a null reference. Some methods throw explicitly this type of exception when they are passed a parameter that is null.

2. ArrayIndexOutOfBoundsException: generated when we access an element of an array using an index that is less than zero, or bigger than the length of the array minus one.

3. IOException: generated by methods that access input/output devices when an error situation occurs.

4. FileNotFoundException: generated when we try to open a non-existent file.

5. NumberFormatException: generated by methods that perform conversions from strings to numbers. For example, Integer.parseInt() generates an exception of this type if the string passed as parameter does not contain a number.