Compile-time Errors

Before we discuss the various errors, it is important to understand that compilers are not very good at recovering from errors. Frequently, a single small mistake will cause the compiler to issue a cascade of messages. For example, in the following code, we forgot to write the closing brace of the method `f`:

```
class MyClass {

  void f() {
    int n = 10;
                    // Error, closing brace is missing
  void g() {
    int m = 20;
  }
}
```

An attempt to compile the program results in *three* error messages:

```
MyClass.java:5: illegal start of expression void g() {
  ^
MyClass.java:8: ';' expected
}
^
MyClass.java:9: '}' expected
^
```

Do not invest any effort in trying to fix multiple error messages! Concentrate on fixing the first error and then recompile.

ECLIPSE is much better at diagnosing this error and produces only one message:

Syntax error, insert "}" to complete MethodBody.

2.1 Syntax errors

Some errors are caused by violations of the syntax of JAVA. Although they are easy to understand, there is no easy way to find the exact cause of such errors except by checking the code around the location of the error *character by character* looking for the syntax error.

2.1.1 ...expected

The syntax of JAVA is very specific about the required punctuation. This error occurs, for example, if you forget a semicolon at the end of a statement or don't balance parentheses:

```
if (i > j   // Error, unbalanced parentheses max = i   //
  Error, missing semicolon
else
  max = j;
```

Unfortunately, this syntax error is not necessarily caught precisely at the point of the mistake so you must carefully check the preceding characters in the line or even in a previous line in order to find the problem. ECLIPSE:

Syntax error, insert ") Statement" to complete IfStatement. Syntax error, insert ";" to complete Statement

ECLIPSE is more informative as to the precise syntax error encountered.

2.1.2 unclosed string literal

String literals must be enclosed in quotation marks.[1] This error occurs if you fail to terminate the literal with quotation marks. Fortunately, the syntax of JAVA requires that a string literal appear entirely on one line so the error message appears on the same line as the mistake. If you need a string literal that is longer than a single line, create two or more literals and concatenate them with +:

```
String longString =
  "This is first half of a long string " + "and this is the
  second half.";
```

ECLIPSE: **String literal is not properly closed by a double-quote**. In ECLIPSE you can write a string literal of arbitrary length and the environment will break the string and insert the + automatically.

[1]A *literal* is a source-code representation of a value; most literals are of primitive types like **int** or

char, but there are also literals of type `String` and the literal **null** of any reference type.

2.1.3   illegal start of expression

Most programming constructs are either statements or expressions. This error occurs when an expression is expected but not found. *In* JAVA, *an assignment statement is considered to be an expression which returns a value*, so errors concerning expressions also apply to assignment statements.[2] Examples:

- An extra right parenthesis after the condition of an **if**-statement: **if** `(i > j) )`   `//`

  ```
  Error, extra parenthesis
    max = i;
  ```

  ECLIPSE: **Syntax error on token ")", delete this token**.[3] ECLIPSE diagnoses this as a simple syntax error and does not mention expressions.

- Forgetting the right-hand side of an assignment statement:

  ```
  max = ;        // Error, missing right-hand side
  ```

  ECLIPSE: **Syntax error on token "=", Expression expected after this token**.

2.1.4   not a statement

This error occurs when a syntactically correct statement does not appear where it should. Examples:

- Writing an assignment statement without the assignment operator:

  ```
  max ;        // Error, missing =
  ```

  ECLIPSE:
  Syntax error, insert "AssignmentOperator Expression" to complete Expression.

- Misspelling **else**:

  ```
  if (i > j) max = i;
  els ;        // Error, else not spelled correctly
  ```

  The reason you do not get "else expected" is that you need not write an **else** alternative so this just looks like a bad statement.

[2]The value of an assignment statement considered as an expression is the value of the expression on the right-hand side that is assigned to the variable on the left-hand side.

[3]The syntax of a programming language is defined in terms of *tokens* consisting of one or more characters. Identifiers and reserved keywords are tokens as are single characters like + and sequences of characters like !=.

ECLIPSE:
els cannot be resolved[4]
Syntax error, insert "AssignmentOperator Expression" to complete Expression.

- The following code:

```
if (i > j) max = i;
els         // Error, else not spelled correctly max = j;
```

results in a weird error message:

```
x.java:6: cannot find symbol symbol   :
class els location: class x
els
```

The reason is that the compiler interprets this as a declaration:

```
els max = j;
```

and can't find a class `els` as the type of the variable `max`. ECLIPSE:
Duplicate local variable max els cannot be resolved to a type.

These messages are more helpful: first, the use of the word *type* instead of *class* is more exact because the type of a variable need not be a class (it could be a primitive type or interface); second, the message about the duplicate variable gives an extra clue as to the source of the error.

- The error can also be caused by attempting to declare a variable whose name is that of a reserved keyword:

```
void f() {
  int default = 10;
}
```

ECLIPSE: **Syntax error on token "default", invalid VariableDeclaratorId**.

[4]The same identifier can be used in different methods and classes. An important task of the compiler is to *resolve* the ambiguity of multiple uses of the same identifer; for example, if a variable is declared both directly within a class and also within a method, the use of its unqualified name is resolved in favor of the local variable. This error message simply means that the compiler could not obtain an (unambiguous) meaning for the identifier `els`.

2.2 Identifiers

2.2.1 cannot find symbol

This is probably the most common compile-time error. All identifiers in JAVA must be declared before being used and an inconsistency between the declaration of an identifier and its use will give rise to this error. **Carefully check the spelling of the identifier.** It is easy to make a mistake by using a lower-case letter instead of an upper case one, or to confuse the letter O with the numeral 0 and the letter l with the numeral 1.

Other sources of this error are: calling a constructor with an incorrect parameter signature, and using an identifier outside its scope, for example, using an identifier declared in a `for`-loop outside the loop:

```java
int[] a = {1, 2, 3};
```

```java
int sum = 0;
for (int i = 0; i < a.length; i++) sum = sum + a[i];
System.out.println("Last = " + i); // Error, i not in scope
```

ECLIPSE: **... cannot be resolved**.

2.2.2 ... is already defined in ...

An identifier can only be declared once in the same scope:

```java
int sum = 0;
double sum = 0.0; // Error, sum already defined
```

ECLIPSE: **Duplicate local variable sum**.

2.2.3 array required but ... found

This error is caused by attempting to index a variable which is not an array.

```java
    int max(int i, int j) {
if (i > j) return i;
 else return j[i];      // Error, j is not an array
}
```

ECLIPSE: **The type of the expression must be an array type but it resolved to int**.

2.2.4 ... has private access in ...

It is illegal to access a variable declared private outside its class. ECLIPSE: **The field .
.. is not visible**.

Computation

This group of errors results from code that is syntactically correct but violates the semantics of JAVA, in particular the rules relating to type checking.

2.1.1 variable ... might not have been initialized

An *instance variable* declared in a class has a default initial value.[5] However, a *local variable* declared within a method does not, so you are required to initialize it before use, either in its declaration or in an assignment statement:

```java
void m(int n) { // n is initialized from the actual  parameter
  int i, j;
  i = 2;         // i is initialized by the assignment int k = 1;      //
  k is initialized in its declaration if (i == n) // OK
   k = j;        // Error, j is not initialized
  else
   j = k;
}
```

The variable must be initialized on *every* path from declaration to use even if the semantics of the

program ensure that the path cannot be taken:

```
void m(int n) {
  int i;
  if (n == n)        // Always true i = n;
  else
    n = i;           // Error, although never  executed!!
}
```

ECLIPSE: **The local variable . . . may not have been initialized**.

Note: If the expression in the `if`-statement can be computed at compile-time:

```
if (true)           // OK
if ('n' == 'n')     // OK
```

the error will not occur.

2.1.2   . . . in . . . cannot be applied to . . .

This error message is very common and results from an incompatibility between a method call and the method's declaration:

[5]A *class* is a template that is used to *create* or *instantiate instances* called *objects*. Since memory is allocated separately for each object, these variables are called *instance variables*.

```
void m(int i) { ... }
```

```
    m(5.5);      // Error, the literal is of type double, not  int
```

2.1.3   incompatible types

JAVA checks that the type of an expression is compatible with the type of the variable in an assignment statement and this error will result if they are incompatible:

```
boolean b = true;
int a = b;          // Error, can't assign boolean to int
```

ECLIPSE: **Type mismatch: cannot convert from boolean to int**.

[6]The *Application Programming Interface (API)* describes how to use the library of classes supplied as part of the JAVA system. By extension, it is also used as a name for the library itself.

2.1.4   possible loss of precision

This error arises when trying to assign a value of higher precision to a variable of lower precision without an explicit type cast. Surprisingly, perhaps, floating point literals are of type `double` and you will get this message if you try to assign one to a variable of type `float`:

```
float sum = 0.0;                // Error, literal is not of type float
```

The correct way to do this is to use a literal of type `float` or an explicit type cast:

```
float sum = 0.0f;          // OK
float sum = (float) 0.0;   // OK
```

ECLIPSE: **Type mismatch: cannot convert from double to float**.

Check the declaration and call carefully. You can also get the error message **cannot find symbol** if the declaration of the method is in the API.[6]

ECLIPSE: **The method ... in the type ... is not applicable for the arguments ....** .

2.1.5 operator ... cannot be applied to ...,...

Operators are only defined for certain types, although implicit type conversion is allowed between certain numeric types:

```
int a = 5;
boolean b = true;
int c = a + b;       // Error, can't add a boolean value
double d = a + 1.4; // OK, int is implicitly converted to  double
```

ECLIPSE: **The operator + is undefined for the argument type(s) int, boolean**.

In the C language it is (unfortunately!) legal to write

```
if (a = b)
```

using the assignment operator instead of

```
if (a == b)
```

using the equality operator. The meaning is to execute the assignment, convert the value to an integer and use its value to make the decision for the `if`-statement (zero is false and non-zero is true). In JAVA, the as- signment is legal and results in a value, but (unless `a` is of type `boolean`!) this error message will result because the expression in an `if`-statement must be of type `boolean`, not `int`. Writing `==` instead of `=` becomes a sim- ple compile-time error in JAVA, whereas in C this error leads to runtime errors that are extremely difficult to find.

2.2.5 inconvertible types

Not every type conversion is legal:

```
boolean b = true;
int x = (int) b; // Error, can't convert boolean to  int
```

ECLIPSE: **Type mismatch: cannot convert from ... to ....** .

2.3   Return statements

There are a number of errors related to the structure of methods and their return statements; in general they are easy to fix.

2.3.1   missing return statement

When a method returns a non-void type, *every* path that leaves the method must have a `return`-statement,[7] *even* if there is no way that the path can be executed:

```
        int max(int i, int j) {
  if (i > j) return i;
  else if (i <= j) return j;
  // Error: what about the path when i>j and i<=j are both  false?!!
}
```

2.3.2 invalid method declaration; return type required

Every method *except constructors* must have a return type or **void** specified; if not, this error will arise:

```
max(int i, int j) {
  ...
}
```

The error frequently occurs because it is easy to misspell the name of a constructor; the compiler then thinks that it is a normal method without a return type:

```
class MyClass { MyClass(int i) { ... }
  Myclass(int i, int j) { ... } // Error because of the lowercase  c
}
```

ECLIPSE: **Return type for the method is missing**.

2.3.3 cannot return a value from method whose result type is void

Conversely, a **return**-statement in a void method must not have an expression:

```
void m(int i, int j) {
        return i + j;          // Error, the method was declared void
}
```

ECLIPSE: **Void methods cannot return a value**.

2.3.4 missing return value

A method returning a type must have a **return**-statement that includes an expression of the correct type:

```
int max(int i, int j) {
  return;                 // Error, missing int expression
}
```

ECLIPSE: **This method must return a result of type int**.

Adding a dummy alternative **else return** 0; at the end of the method will enable successful compilation. ECLIPSE: **This method must return a result of type int**. This ECLIPSE message is rather hard to understand because, clearly, the method does return a result of type **int**, just not on all paths.

2.3.5 unreachable statement

The error can occur if you write a statement *after* a return statement:

```
void m(int j) { System.out.println("Value is " + j);
  return;
  j++;
}
```

The check is purely syntactic, so the error *will occur* in the following method:

```
if (true) {
```

```
      return n + 1;   // Only this alternative executed, but ...
  }
  else {
    return n - 1;
    n = n + 1;      // ... this is an error
  }
```

ECLIPSE: **Unreachable code**.

2.4 Access to static entities

The modifier `static` means that a variable or method is associated with a *class* and not with individual *objects* of a class.[8] Normally, static entities are rarely used in JAVA (other than for the declaration of constants), because programs are written as classes to be instantiated to create at least one object:

```
class MyClass {

  int field;
  void m(int parm) { field = parm;
  }
  public static void main(String[] args) {
    MyClass myclass = new MyClass();   // Create object myclass.m(5);  // Call
      object's method System.out.println(myclass.field); // Access object's field

  }
}
```

Some teachers of elementary programming in JAVA prefer to start with a procedural approach that involves writing a class containing static variables and static methods that are accessed from the main method without instantiating an object as was done above:

[8]`static` has other uses that we do not consider here: (a) as a modifier for nested classes and (b) in static initializers.

If the variable is declared directly within its class, you should initialize it either at its declaration or within the constructor:

```
class YourClass {
  MyClass[] my;                  // Can't initialize here because ...

  YourClass(int size) {     // ... size of array different for each object my = new
  MyClass[size];
  }
}
```

The exception is likely to occur when you declare an array whose elements are of reference type. It is easy to forget that the array contains only references and that each element must be separately initialized:

```
MyClass[] my = new MyClass[4]; // Array of references my[1].m(5);      //
Raises an exception

for (int i = 0; i < my.length; i++)
  my[i] = new MyClass();        // Instantiate objects my[1].m(5);      //
OK
```

[3]The name of the exception is an anachronism, because there are no (explicit) *pointers* in JAVA.

```
class MyClass1 {
 static int field;
 static void m(int parm) {
   field = parm;
 }
 public static void main(String[] args) {
   m(5);                           // OK
   System.out.println(field);   // OK
 }
}
```

2.4.1 non-static variable . . . cannot be referenced from a static context

Since the method `main` is (required to be) static, so must any variable declared in the class that is accessed by the method. Omitting the modifier results in a compile-time error:

```
int field;                    // Forgot "static"
  ...
System.out.println(field); // Error, which field?
```

The variable `field` does not exist until an object of the class is instantiated, so using the identifier `field` by itself is impossible before objects are instantiated. Furthermore, it is ambiguous afterwards, as there may be many objects of the same class. ECLIPSE: **Cannot make a static reference to the non-static field . . . .**

2.4.2   non-static method cannot be referenced from a static context

Similarly, a non-static method cannot be called from a static method like `main`; the reason is a bit subtle. When a non-static method like `m` is executed, it receives as an *implicit* parameter the reference to an object. (The reference can be explicitly referred to using **this**.) Therefore, when it accesses variables declared in the class like `field`:

```
void m(int parm) { // Forgot "static"
  field = parm;      // Error, which  field?
}

public static void main(String[] args) {
  m(5);
}
```

it is clear that the variable is the one associated with the object referenced by **this**. Thus, in the absence of an object, it is meaningless to call a non-static method from a static method.
ECLIPSE:
Cannot make a static reference to the non-static method . . . from the type . . .

Runtime Errors

When the JAVA interpreter encounters an error during runtime it *throws an exception* and prints a *stack trace* showing the entire call stack—the list of methods called from the main program until the statement that caused the exception.[1] Consider the fol- lowing program:

```java
class Test {

  public static void main(String[] args) { String s = "Hello
    world"; System.out.println(s.substring(10,12));
  }
}
```

We are trying to extract a substring of the string `s` but the upper index `12` is not within the string. Attempting to execute this code will cause an exception to be thrown:

```
Exception in thread "main"
  java.lang.StringIndexOutOfBoundsException:
  String index out of range: 12
      at java.lang.String.substring(Unknown Source) at
      Test.main(Test.java:4)
```

The exception was thrown from *within* the library class `String` during the execution of the method `substring`. This method was called at line `4` of method `main` in the class `Test`. It is unlikely that there is an error that causes an exception in a well established class of the JAVA library like `String`; more likely, the `String` class itself identified the error and threw the exception explicitly, so we are advised to seek the error starting at the *deepest* method of our own program.

The calls in the stack trace are listed in reverse order of invocation, that is, the first call listed is the deepest method (where the exception occurred), while the last call listed is the shallowest method, namely `main`.

---

[1]Many programming languages prefer the term *raises* an exception and even *The Java Language Spec- ification* uses this term on occasion.

Exceptions can be *caught* by writing a block of code called an *exception handler* that will be executed when an exception is thrown.[2] Exception handlers are discussed in textbooks on the JAVA language; here we limit ourselves to explaining the exceptions that indicate programming errors. We further limit ourselves to exceptions com- monly thrown by the language core; classes in the API define their own exceptions and these are explained in the API documentation.

3.1 Out of range

These exceptions are the most common and are caused by attempting to access an array or string with an index that is outside the limits of the array or string. C pro- grammers will be familiar with the difficult bugs that are caused by such errors which "smear" memory belonging to other variables; in JAVA the errors cause an exception immediately when they occur thus facilitating debugging.

3.1.1 ArrayIndexOutOfRange

This exception is thrown when attempting the index an array `a[i]` where the index `i` is not within the values `0` and `a.length-1`, inclusive. Zero-based arrays can be con- fusing; since the `length` of the array is larger than the last index, it is not unusual to write the following in a **`for`**-statement by mistake:

```
final static int SIZE = 10;
```

```
int a = new int[SIZE];
```

```
for (int i = 0; i <= SIZE; i++)     // Error, <= should be <
```

```
for (int i = 0; i <= a.length; i++) // Better, but still an  error
```

3.1.2 StringIndexOutOfRange

This exception is thrown by many methods of the class `String` if an index is not within the values `0` and `s.length()`, inclusive. Note that `s.length()` is valid and means the position just after the last character in the string.

3.2   Dereferencing null

3.2.1   NullPointerException

A variable in JAVA is either of a *primitive type* such as **`int`** or **`boolean`** or of a *reference type*: an array type, the type of a class defined in the JAVA API such as `String`, or the type of a class that you define such as `MyClass`. When declaring a variable of a reference type, you only get a variable that can hold a reference

to an object of that class. At this point, attempting to access a field or method of the class will cause the exception `NullPointerException` to be thrown:[3]

[2]An alternate terminology is to say that an exception is *handled* by an exception handler.

```
public static void main(String[] args) {}
  MyClass my;              // Can hold a reference but doesn't yet
  my.m(5);                 // Throws an exception
}
```

Only after instantiating the class and executing the constructor do you get an object:

```
public static void main(String[] args) {}
  MyClass my;              // Can hold a reference but doesn't yet
  my = new MyClass();      // Instantiates an object and assigns reference my.m(5);   // OK
}
```

This exception is not often seen in this context because JAVA style prefers that the declaration of the variable include the instantiation as its initial value:

```
public static void main(String[] args) {}
  MyClass my = new MyClass(); // Declaration + instantiation my.m(5);   // OK
}
```

Finally, `NullPointerException` will occur if you get a reference as a return value from a method and don't know or don't check that the value is non-null:[4]

```
Node node = getNextNode();
if (node.key > this.key) ...   // Error if node is null!

if (node != null) {            // This should be done first
  if (node.key > this.key) ...
}
```

3.3   Computation

The following three exceptions occur when you try to convert a string to a number and the form of the string does not match that of a number, for example `"12a3"`.

3.3.1   InputMismatchException

This exception is thrown by the class `Scanner`, which is a class introduced into version 5 of JAVA to simplify character-based input to programs.

3.3.2   IllegalFormatException

This exception is thrown by the method `format` in class `String` that enables output using format specifiers as in the C language.

3.3.3   NumberFormatException

This exception is thrown by methods declared in the numeric "wrapper" classes such as `Integer` and `Double`, in particular by the methods `parseInt` and `parseDouble` that convert from strings to the primitive numeric types.

3.3.4   ArithmeticException

This exception is thrown if you attempt to divide by zero.

---

[4]You could *know* this if the method has been verified for a *postcondition* that specifies a non-null return value.

Important note

Most computational errors *do not* result in the raising of an exception! Instead, the result is an artificial value called $\texttt{NaN}$, short for *Not a Number*. This value can even be printed:

```
double x = Math.sqrt(-1); // Does not throw an exception! System.out.println(x);
```

Any attempt to perform further computation with $\texttt{NaN}$ does not change the value. That is, if $\texttt{x}$ is $\texttt{NaN}$ then so is $\texttt{y}$ after executing $\texttt{y = x+5}$. It follows that an output value of $\texttt{NaN}$ gives no information as to the statement that first produced it. You will have to set breakpoints or use print statements to search for it.

3.4   Insufficient memory

Modern computers have very large memories so you are unlikely to encounter these exceptions in routine programming. Nevertheless, they can occur as a side effect of other mistakes.

3.4.1   outOfMemoryError

This exception can be thrown if you run out of memory:

```
int a = new int[100000000];
```

3.4.2   StackOverFlowArea

A stack is used to store the activation record of each method that is called; it contains the parameters and the local variables as well as other information such as the return address. Unbounded recursion can cause the Java Virtual Machine to run out of space in the stack:

```
int factorial(int n) {

 if (n == 0) return 1;
 else return n * factorial(n + 1); // Error, you meant n - 1
}
```

3.5   Program execution

You can run a JAVA program from within an environment or by executing the inter- preter from the command line:

```
java Test
```

where `Test` is the name of a class containing a `main` method.

Suggestion: It is convenient to write `main` methods in most classes and to use then for testing classes individually. After the classes are integrated into a single program, you only need ensure that the interpreter is invoked on the class that contains the "real" `main` method.

Two runtime errors can occur if the interpreter is not successful in finding and run- ning the program.

3.5.1   NoClassDefFoundError

The interpreter must be able to find the file containing a class with the `main` method, for example, `Test.class`. If packages are not used, this must be in the directory where the interpreter is executed. Check that the name of the class is the same as the name of the file without the extension. Case is significant!

Warning! If you are compiling and running JAVA from a command window or shell with a history facility, you are likely to encounter this error. First, you compile the program:

```
javac MyClass.java
```

and then you recall that line and erase the `c` from `javac` to obtain the command for the interpreter:

```
java MyClass.java
```

Unfortunately, `MyClass.java` is *not* the name of the class so you will get this excep- tion. You must erase the `.java` extension as well to run the interpreter:

```
java MyClass
```

If you are using packages, the main class must be in a *subdirectory* of that name. For example, given:

```
package project1.userinterface;

class Test {
        public static void main(String[] args) {
         ...
  }
}
```

the file `Test.`**`class`** must be located in the directory `userinterface` that is a subdi- rectory of `project1` that is a subdirectory of the directory where the interpreter is executed:

```
c:\projects> dir
   <DIR> project1 c:\projects> dir
project1
   <DIR> userinterface
c:\projects> dir project1\userinterface Test.class
```

The program is invoked by giving the fully qualified name made up of the package names and the class name:

```
c:\projects> java project1.userinterface.Test
```

3.5.2 NoSuchMethodFoundError: main

This error will occur if there is no method `main` in the class, or if the declaration is not precisely correct: the **`static`** modifier, the **`void`** return type, the method name `main` written in lower case, and one parameter of type `String[]`. If you forget the **`public`** modifier, the error message is **Main method not public**.

5.1 Forcing a stack trace

Suppose that you have isolated a bug to a certain method but you do not know which call of that method was responsible. You can force the interpreter to print a trace of the call stack by inserting the line:

```
new Exception().printStackTrace();
```

within the method. It creates a new object of type `Exception` and then invokes the method `printStackTrace`. Since the exception is not thrown, the execution of the program proceeds with no interruption. A program can always be terminated by calling `System.exit(n)` for an integer `n`.

5.2 Leave the debug statements in the program

Once you have found a bug, it is tempting to delete the print statements used for debugging, but it is better not to do so because you may need them in the future. You can comment out the statements that you don't need, but a better solution is to declare a global constant and then use it to turn the print statements on and off:

```
public class Global {
  public static boolean DEBUG = false; public static void
  print(int[] a) {...} public static void print(Object[] a)
  {...}
}
if (Global.DEBUG) Global.print(nodes);
```

Assignment and equality

JAVA programmers make mistakes in the use of the assignment and equality oper- ators, especially when strings are used. The concept of reference semantics is (or should be) explained in detail in your textbook, so we limit ourselves to a reminder of the potential problems.

4.1   String equality

Given:

```
String s1 = "abcdef"; String s2 = "abcdef";
String s3 = "abc" + "def"; String s4 =
"abcdef" + ""; String s5 = s1 + ""; String t1
= "abc";
String t2 = "def"; String s6 = t1 + t2;
```

all strings `sn` are equal when compared using the method `equals` in the class `String` that compares the *contents* pointed to by the reference:

```
if (s1.equals(s5))  // Condition evaluates to true
```

The string *literal* `"abcdef"` is stored only once, so strings `s1`, `s2` and (perhaps surpris- ingly) `s3` and `s4` are also equal when compared using the equality operator `==` that compares the references themselves:

```
if (s1 == s3)       // Condition evaluates to true
```

However, `s5` and `s6` are not equal (==) to `s1` through `s4`, because their values are created at runtime and stored separately; therefore, their references are not the same as they are for the literals created at compile-time.

Always use `equals` rather than `==` to compare strings, unless you can explain why the latter is needed!

4.2   Assignment of references

The assignment operator copies references, not the contents of an object. Given:

```
int[] a1 = { 1, 2, 3, 4, 5 };
int[] a2 = a1; a1[0] = 6;
```

since `a2` points to the same array, the value of `a2[0]` is also changed to 6. To copy an array, you have to write an explicit loop and copy the elements one by one.

To copy an object pointed to by a reference, you can create a new object and pass the old object as a parameter to a constructor:

```
class MyClass {
  int x;
  MyClass(int y) { x = y; }
  MyClass(MyClass myclass) { this.x = myclass.x; }
}

class Test {
  public static void main(String[] args) { MyClass myclass1 =
    new MyClass(5); MyClass myclass2 = new MyClass(myclass1);
```

```
   myclass1.x = 6;
   System.out.println(myclass1.x); // Prints 6 System.out.println(myclass2.x); // Prints  5
 }
}
```

Alternatively, you can use `clone` as described in JAVA textbooks.

On debugging without a debugger

Debugging is perhaps best done using a debugger such as those provided by inte- grated development environments. Nevertheless, many programmers debug pro- grams by inserting print statements. This section describes some of the techniques that can be used in JAVA.

5.1 Printing number data and arrays

The methods `System.out.print` and `System.out.println` are predefined for primitive types as well as for strings:

```
int i = 1;
double d = 5.2; System.out.print(i);
System.out.println(d);
```

Furthermore, automatic conversion to `String` type is performed by the concatenation operator `+`:

```
System.out.println("d = " + d + "and i = " + i);
```

The print statements can not print an entire array, so a method must be written:

```
public static void print(int[] a) {
  for (int i = 0; i < a.length; i++) System.out.print(a[i]);
  System.out.println();
}

public static void main(String[] args) {
  int[] a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  print(a);
}
```

Since the number of elements of an array can be large, it is better to write the method so that it inserts a newline after printing a fixed number of elements:

```
public static void print(int[] a) {
  for (int i = 0; i < a.length; i++) {
    System.out.print(a[i]);
    if (i % 8 == 7) System.out.println();
  }
  System.out.println();
}
```

You can put this static method in a publicly accessible class and use it to print any integer array. Similar methods can be written for the other primitive types.

5.2 Converting objects to strings

Within the class `Object`, the root class for all other classes, a method `toString` is de- clared. The default implementation will not give useful information, so it is a good idea to override it in each class that you write:

```
class Node { int key; double
  value;
  public String toString() {
    return "The value at key " + key + " is " + value;
  }
}
```

Then, you can simply call the print statements for any object of this class and the conversion to `String` is done automatically:

```
Node node = new Node();
...
System.out.println(node);
```

The predefined class `java.util.Arrays` constains a lot of useful (static) methods for working with arrays, among them `toString` methods for arrays whose elements are of any primitive type:

```
int[] a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
System.out.println(java.util.Arrays.toString(a));
```

or

```
import java.util.Arrays;
   ...
System.out.println(Arrays.toString(a));
```

You will receive a predefined representation of the array (the elements are separated by commas), so if you want an alternate representation (such as printing on multiple lines) you will have to write your own print method as we did in the previous section.

An array of objects of *any reference type* can be printed by defing a single method since any object can be converted to `Object`:

```
public static void print(Object[] a) {
  for (int i = 0; i < a.length; i++) System.out.print(a[i]);
  System.out.println();
}

Node[] nodes = new Node[];
   ...                     // Create elements of the array print(nodes);
```

or by calling the predefined method `deepToString`:

```
Node[] nodes = new Node[];
   ...                     // Create elements of the array
System.out.println(java.util.Arrays.deepToString(node));
```