Introduction to Java

The Java architecture consists of:

- a high-level object-oriented programming language,
- a platform-independent representation of a compiled class,
- a pre-defined set of run-time libraries,
- a virtual machine.

This book is mainly concerned with the language aspects of Java and the associated `java.lang` library package. Consequently, the remainder of this section provides a brief introduction to the language. Issues associated with the other components will be introduced as and when needed in the relevant chapters.

The introduction is broken down into the following components

- identifiers and primitive data types
- structured data types
- reference types
- blocks and exception handling
- control structures
- procedures and functions
- object oriented programming, packages and classes
- inheritance
- interfaces
- inner classes.

1 Identifiers and primitive data types

Identifiers

Java does not restrict the lengths of identifiers. Although the language does allow the use of a “_” to be included in identifier names, the emerging style is to use a mixture of upper and lower case characters. The following are example identifiers:

```
exampleNameInJava
example_name_in_Java
```

Note that Java is case sensitive. Hence, the identifier `exampleNameInJava` is different from `ExampleNameInJava.`

Primitive data types

Java provides both a variety of discrete data types and a floating point type.

Discrete data types. The following discrete data types are supported:

- `int` — a 32-bit signed integer;
- `short` — a 16-bit signed integer;
- `long` — a 64-bit signed integer;
- `byte` — an 8-bit signed integer;
- `boolean` — the truth values, `true` and `false`;
- `char` — Unicode characters (16 bits).

All the usual operators for these types are available.

Enumeration types have been introduces as of Java 1.5. Although not a primitive type it behaves like a descrete type and can be used in a switch statement. The following example illustrates a simple enumeration type for days of the week.

```
public enum Day{SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                 THURSDAY, FRIDAY, SATURDAY}
```

Floating point numbers

Java provides `float` and `double` types that support the 32 and 64 bit IEEE 754 floating point values respectively. Note, that floating point literals are automatically considered to be of double precision. Consequently, they must either be explicitly converted to `float` or they can be followed by the letter `f`. For example:

```
float bodyTemperature = (float) 98.6;
// or
float bodyTemperature = 98.6f;
```

Note that the assignment operator is “=” and that comments are delimited by “/*” and “*/”. Java also allows “//” for comments ending a line.

2 Structured data types

Java supports arrays; both single and multi dimensional arrays can be created; for example:

```
final int max = 10; // a constant
float[] reading = new float[max]; // index is 0 .. max-1
boolean[][] switches = new boolean[max][max];
```

Note that arrays are represented by objects and that indexes start at 0. The length of the array can be determined by the length field associated with the array

Java has no record structure as such. However, the same effect can be achieved using classes. The details of classes will be given in Section 7 but, for now, consider the following class for the date:

```
class Date {
  int day, month, year;
}
Date birthDate = new Date();
birthDate.day = 31;
birthDate.month = 1;
birthDate.year = 2000;
```

It is not possible to express range constraints on the values of the date's components. Note also that as a “record” is a class, an allocator (the `new` operator) must be used to create an instance of the `Date`. Initialization of the object can be achieved using constructors (see Section 7).

Important note

All objects created by the `new` operator are stored in an area of memory called the heap. An activity called *garbage collection* will free up any associated memory when the object is no longer referenced.

3 Reference types

All objects in Java are references to the actual locations containing the encapsulated data, hence no additional access or pointer type is required. Furthermore, no forward declaration is required. For example:

```
class Node {
  int value;
  Node next;
     // The type node can be used even though
     // its declaration has not been completed yet.
}
```

Note that as a result of representing objects as reference values, comparing two objects is a comparison between their references **not** their values. Hence,

```
Node ref1 = new Node();
Node ref2 = new Node();
. . .
if(ref1 == ref2) { ... }
```

will compare the locations of the objects not the values encapsulated by the objects (in this case, the values of the `value` and `next` fields). To compare values requires a class to implement an explicit `equals` method. A similar situation occurs with object assignment. The assignment operator assigns to the reference. To create an actual copy of an object requires the class to provide methods to `clone` the object.

4 Blocks and exception handling

In Java, a block (or compound statement) is delimited by “{“ and “}”, and usually has the following structure

```
{
  < declarative part >

  < sequence of statements >
}
```

although the declarative part can be dispersed throughout the block.

Java can have exception handlers at the end of a block if the block is labeled as a `try` block. Each handler is specified using a catch statement. Consider, the following:

```
try {
 //code that might throw an exception
} catch (Exception err) {
  // Exception caught, print error message on
  // the standard output.
  System.out.println(err.getMessage());
}
```

The catch statement is like a function declaration, the parameter of which identifies the exception type to be caught. Inside the handler, the object name behaves like a local variable. A handler with parameter type `T` will catch a thrown object of type `E` if:

- `T` and `E` are the same type,
- or `T` is a parent (super) class of `E` at the throw point.

It is this last point that integrates the exception handling facility with the object-oriented programming model. In the above example the `Exception` class is the root class of all application exceptions. Hence, the catch clause will catch all application exceptions. Here, `getMessage` is a method declared in the `Exception` class. A call to `E.getMessage` will execute the appropriate routine for the type of object thrown.

If no exception handler is found in the calling context of a function, the calling context is terminated and a handler is sought in its calling context. Hence, Java supports exception propagation.

Finally clauses

Java also supports a **`finally`** clause as part of a try statement. The code attached to this clause is guaranteed to execute whatever happens in the try statement irrespective of whether exceptions are thrown, caught, propagated or, indeed, even if there are no exceptions thrown at all.

```
try {
   ...
} catch(...) {
   ...
} finally {
  // code that will be executed under all circumstances
}
```

Checked and unchecked exceptions

In Java, all exceptions are subclasses of the predefined class `java.lang.Throwable`. The language also defines other classes, for example: `Error`, `Exception`, and `RuntimeException`. The relationship between these (and some common exceptions) is depicted in Figure 1.

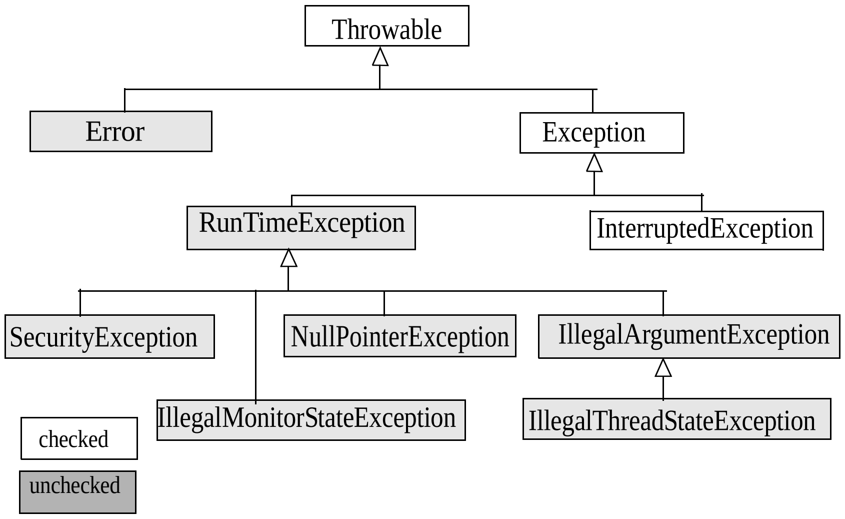

FIGURE 1. **Part of the The Java Predefined Throwable Class Hierarchy**

Throughout this book, the term Java exception is used to denote any class derived from `Exception`. Objects derived from `Error` describe internal errors and resource exhaustion in the Java run-time support system. Although these errors clearly have a major impact on the program, there is little that the program can do when they are thrown (raised) as no assumptions can be made concerning the integrity of the system. Objects derived from the `Exception` hierarchy represent errors that programs can handle or can throw themselves. `RuntimeException`s are those exceptions that are raised by the run-time system as a result of a program error. They include errors such as those resulting from a bad cast (`ClassCastException`), array bounds error (`IndexOutOfBoundException`), a null pointer access (`NullPointerException`), integer divide by zero (`ArithmeticException`) etc.

Throwable objects which are derived from `Error` or `RuntimeExceptions` are called **unchecked** exceptions, the others are termed **checked** exceptions. Checked exceptions must be declared by the function that can throw them. The compiler will check that an appropriate handler can be found. The compiler makes no attempt to ensure that handlers for unchecked exceptions exist.

5 Control structures

Control structures can be grouped together into three categories: sequences, decisions and loops.

Sequence structures

Most languages implicitly require sequential execution, and no specific control structure is provided. The definitions of a block in Java indicate that between `{` and `}` there is a sequence of statements. Execution is required to follow this sequence.

Decision structures

The most common form of decision structure is the `if` statement; for example:

```
if(A != 0) {
  if(B/A > 10) {
    high = 1;
  } else {
    high = 0;
  }
}
```

In general, a multiway decision can be more explicitly stated and efficiently implemented, using a `switch` structure.

```
switch(command) {
  case 'A'   :
  case 'a'   : action1(); break;    /* A or a */
  case 't'   : action2(); break;
  case 'e'   : action3(); break;
  case 'x'   :
  case 'y'   :
  case 'z'   : action4(); break;    /* x, y or z */
  default    :                      /* no action */
}
```

As with C and C++, it is necessary to insert statements to break out of the switch once the required command has been identified. Without these, control continues to the next option (as with the case of A).

Loop (iteration) structures

Java supports the usual `for` and `while` statements. The following example illustrates the `for` statement; the code assigns into the first ten elements of array, A, the value of their positions in the array:

```
for(i = 0; i <= 9; i++) {
             /* i must be previously declared */
  A[i]= i;   /* i can be read/written in the loop */
}            /* the value of i is defined */
             /* after the loop */
```

The free use of the loop control variable (`i` here) in this manner can be the cause of many errors, consequently, Java also allows a local variable to be declared inside the `for` loop.

```
for(int i = 0; i <= max; i++) {
  A[i]= i;
}
```

An example `while` statement is given below:

```
while(<expression>) {
  /* Expression evaluating to true or false. */
  /* False implies loop termination. */
  <sequence of statements>
}
```

Java also supports a variant where the test occurs at the end of the loop:

```
do {
  < sequence of statements>
} while (<expression>);
```

The flexibility of an iteration statement is increased by allowing control to pass out of the loop (that is, the loop to terminate) from any point within it (`break`) or for control to pass directly to the next iteration (`continue`):

```
while(true) {
  ...
  if(<expression1>) break;
  if(<expression2>) continue;
  ...
}
```

6 Procedures and functions

Procedures and functions can only be declared in the context of a class, they are known collectively as **methods**. In fact, strictly Java only supports functions; a procedure is considered to be a function with no return value (indicated by `void` in the function definition).

Java passes primitive data type (int, boolean, float etc.) parameters (often called arguments) by value. Variables of class type are reference variables. Hence, when they are passed as arguments they are copied, but the effect is as if the object was passed by reference. Consider a function that calculates the roots of a quadratic equation:

```
public class Roots {
  double R1, R2;
}

// the following must be declared in a class context
boolean quadratic(double A, double B, double C,
                  Roots R);
```

Note Java requires the roots of the equation to be passed as a class type, as primitive types (including double) are passed by copy and there are no pointer types.

Function bodies

The bodies of functions are illustrated by completing the `quadratic` definitions given above.

```
boolean quadratic(double A, double B,
                  double C, Roots R) {
  double disc;
  disc = B*B - 4.0*A*C;
  if(disc < 0.0 || A == 0.0) { // no roots
    R.R1 = 0;     //  arbitrary values
    R.R2 = 0;
    return false;
  }
  R.R1 = (-B + Math.sqrt(disc)) / (2.0*A);
  R.R2 = (-B - Math.sqrt(disc)) / (2.0*A);
  return true;
}
```

The invoking of a function merely involves naming the function (including any object or class name) and giving the appropriately typed parameters in parentheses.

7 Object-oriented programming, packages and classes

Objects have four important properties [Wegner, 1987]. They have

- **inheritance** (type extensibility)

- **automatic object initialization** (constructors)
- **automatic object finalization** (destructors)
- **dynamic binding of method calls** (polymorphism — sometimes called run-time dispatching of operations).

All of which are supported, in some form, by Java's class and interface mechanisms. Related classes and interfaces can be grouped together into larger units called p**ackages**.

Inheritance is perhaps the most significant concept in an object abstraction. This enables a type (class) to be defined as an extension of a previously defined type. The new type inherits the "base" type but may include new fields and new operations. Once the type has been extended then run-time dispatching of operations is required to ensure the appropriate operation is called for a particular instance of the family of types.

Earlier in this section, a simple class for the date type was introduced.

```
class Date {
  int day, month, year;
}
```

This example only illustrates how data items can be grouped together. The full facilities of a class allow the data items (or instance variables or **fields** as Java calls them) to be encapsulated and, hence, abstract data types to be defined. As an example, consider the class for a queue abstraction.

First, a package to contain the queue can be declared (if there is no named package, then the system assumes an unnamed package). Items from other packages can be imported. Then classes to be declared inside the package are given. One of these classes `Queue` is declared as `public`, and it is only this that can be accessed outside the package. The keyword `public` is termed a **modifier** in Java, the other ones for classes include `abstract` and `final`. An `abstract` class is one from which no objects can be created. Hence, the class has to be extended (to produce a subclass) and that subclass made non-abstract before objects can exist. A `final` modifier indicates that the class cannot be subclassed. No modifier indicates that the class is only accessible within the package. A class can have more than one modifier but certain combinations, such as `abstract` and `final`, are meaningless.

Each class can declare local instance variables (or fields), constructor methods and ordinary (member) methods. A static field is a class-wide field and is shared between all instances of the owning class. The field is accessible even if there are no instances of the class. Constructor methods have the same name as their associated class. An appropriate constructor method is automatically called when objects of the class are created. All methods of the object can also have associated modifiers. These dictate the accessibility of the method; `public`, `protected` and `private` are allowed. Public methods allow full access, `protected` allows access only from within the same package or from with a subclass of the defining class and `private` allows access only from within the defining class. Instance variables can also have these modifiers. If no modifier is given, the access is restricted to the package. Static methods (class-wide methods) can be invoked without reference to an object. They, therefore, can only access static fields or other static methods.

Member methods and instance variables can have other modifiers which will be introduced throughout this book.

The code for the `Queue` class can now be given.

```
package queues; // package name
import somepackage.Element; // import element type

class QueueNode { // class local to package
  Element data; // queued data item
  QueueNode next; // reference to next QueueNode
}

public class Queue {
   // class available from outside the package

  public Queue() { // public constructor
    front = null;
    back = null;
  }

  public void insert(Element E) { //  visible method
    QueueNode newNode = new QueueNode();
    newNode.data = E;
    newNode.next = null;
    if(empty()) {
      front = newNode;
    } else {
      back.next = newNode;
    }
    back = newNode;
  }
  public Element remove() { //visible method
    if(!empty()) {
      Element tmpE = front.data;
      front = front.next;
      if(empty()) back = null;
      return tmpE;
    }
    // Garbage collection will free up the QueueNode
    // object which is now dangling.
    return null; // queue empty
  }
  public boolean empty() { // visible method
    return (front == null);
  }
  private QueueNode front, back; // instance variables
}
```

Java 1.5 has introduced extra facilities for generic programming. The above example assumed that the queue was for the `Element` class. To generalise this class so that it deals with any element, the Element is defined as a generic parameter:

```
package queues; // package name
class QueueNode<Element> { // class local to package
  Element data; // generic queued data item
  QueueNode next; // reference to next QueueNode
}
public class Queue<Element> {
   // class available from outside the package
  public Queue() { // public constructor
    ... // as before
  }
  .. // as before
}
```

The queue can now be instantiated for a particular class.

```
Queue<String> dictionary = new Queue<String>();
```

8 Inheritance

Inheritance in Java is obtained by deriving one class from another. Multiple inheritance is not supported although similar effects can be achieved using interfaces (see below).

Consider an example of a class which describes a two dimensional coordinate system.

```
package coordinates;
public class TwoDimensionalCoordinate {
  public TwoDimensionalCoordinate(
        float initialX, float initialY) { // constructor
    X = initialX;
    Y = initialY;
  }
  public void set(float F1, float F2) {
    X = F1;
    Y = F2;
  }
  public float getX() {
    return X;
  }
  public float getY() {
    return Y;
  }
  private float X;
  private float Y;
}
```

This class can now be extended to produce a new class by naming the base class in the declaration of the derived class with the `extends` keyword. The new class is placed in the same package[1]

```
package coordinates;
public class ThreeDimensionalCoordinate
       extends TwoDimensionalCoordinate {
  // subclass of TwoDimensionalCoordinate
  float Z; // new field
  public ThreeDimensionalCoordinate(
         float initialX, float initialY,
         float initialZ) { // constructor
    super(initialX, initialY);
         // call superclass constructor
    Z = initialZ;
  }
  public void set(float F1, float F2, float F3) {
               //overloaded method
    set(F1, F2); // call superclass set
    Z = F3;
  }
  public float getZ() { // new method
    return Z;
  }
}
```

A new field `Z` has been added and the constructor class has been defined, the `set` function has been overridden, and a new operation provided. Here, the constructor calls the base class constructor (via the `super` keyword) and then initializes the final dimension. Similarly, the overloaded `set` function calls the base class.

All method calls in Java are potentially dispatching (dynamically bound). For example, consider again the above example:

1. In Java, public classes must reside in their own file. Hence, a package can be distributed across one or more files, and can be continually augmented.

```
package coordinates;
public class TwoDimensionalCoordinate {
  // as before
  public void plot() {
   // plot a two dimensional point
  }
}
```

Here the method `plot` has been added. Now if `plot` is overridden in a child (sub) class:

```
package coordinates;
public class ThreeDimensionalCoordinate
       extends TwoDimensionalCoordinate {
  // as before
  public void plot() {
    // plot a three dimensional point
  }
}
```

Then the following:

```
{
  TwoDimensionalCoordinate A = new
                     TwoDimensionalCoordinate(0f, 0f);
  A.plot();
}
```

would plot a two dimensional coordinate; where as

```
{
  TwoDimensionalCoordinate A =
                 new TwoDimensionalCoordinate(0f, 0f);
  ThreeDimensionalCoordinate B =
           new ThreeDimensionalCoordinate(0f, 0f, 0f);
  A = B;
  A.plot();
}
```

would plot a three dimensional coordinate even though A was originally declared to be of type TwoDimensionalCoordinate. This is because A and B are reference types. By assigning B to A, only the reference has changed not the object itself.

The Object class

All classes, in Java, are implicit subclasses of a root class called Object. The definition of this class is given below:

```
package java.lang;
public class Object {
  public final Class getClass();
  public String toString();
  public boolean equals(Object obj);
  public int hashCode();
  protected Object clone()
    throws CloneNotSupportedException;
  public final void wait()
    throws InterruptedException;
    // throws unchecked IllegalMonitorStateException
  public final void wait(long millis)
    throws InterruptedException;
    // throws unchecked IllegalMonitorStateException
  public final void wait(long millis, int nanos)
    throws InterruptedException;
    // throws unchecked IllegalMonitorStateException
  public final void notify()
    // throws unchecked IllegalMonitorStateException;
  public final void notifyAll()
    // throws unchecked IllegalMonitorStateException;
  protected void finalize() throws Throwable;
}
```

There are six methods in the `Object` class which are of interest in this book. The three `wait` and two `notify` methods are used for concurrency control and will be considered in Chapter 3.

The sixth method is the `finalize` method. It is this method that gets called just before the object is destroyed. Hence by overriding this method, a child class can provide finalization for objects which are created from it. Of course, when a class overrides the `finalize` method, its last action should be to call the `finalize` method of its parent. However, it should be noted that `finalize` is only called when garbage collection is about to take place, and this may be some time after the object is no longer in use; there are no explicit destructor methods as in, say, C++. Furthermore, garbage collection will not necessarily take place before a program terminates. In the `System` class there are two methods that can be used to request finalization:

```
package java.lang;
public class System {
  ...
  public static void runFinalization();
  public static void runFinalizeOnExit(boolean value);
     // deprecated
}
```

The first of these methods requests the JVM to complete all outstanding finalization code. The second method (when called with a `true` parameter) indicates that all outstanding finalization be completed before the program exits; however, it has now been deprecated as its use was found to lead to potential deadlocks.

9 Interfaces

Interfaces augment classes to increase the reusability of code. An interface defines a reference type which contains a set of methods and constants. The methods are by definition abstract, so no instances of interfaces can be constructed. Instead, one or more classes can implement an interface, and objects implementing interfaces can be passed as arguments to methods by defining the parameter to be of the interface type. What interfaces do, in effect, is to allow relationships to be constructed between classes outside of the class hierarchy.

Consider, a “generic” algorithm to sort arrays of objects. What is common about all classes that can be sorted is that they support a less than, `<`, or greater than, `>`, operator. Consequently, this feature is encapsulated in an interface. The `Ordered` interface defines a single method `lessThan` that takes as an argument an object whose class implements the `Ordered` interface. Any class which implements the `Ordered` interface must compare its object’s ordering with the argument passed and return whether it is less than the object or not.

```
package interfaceExamples;

public interface Ordered {
  boolean lessThan (Ordered O);
}
```

The class for complex numbers is such a class.

```
import interfaceExamples.*;
class ComplexNumber implements Ordered {
  // the class implements the Ordered interface

  public boolean lessThan(Ordered O) {
     // the interface implementation
    ComplexNumber CN = (ComplexNumber) O; // cast O

    if((realPart*realPart + imagPart*imagPart) <
        (CN.getReal()*CN.getReal() +
         CN.getImag()*CN.getImag())) {
      return true;
    }
    return false;
  }
  public ComplexNumber (float I, float J) {
         // constructor
    realPart = I;
    imagPart = J;
  }
  public float getReal() { return realPart; }
  public float getImag() { return imagPart; }
  protected float realPart;
  protected float imagPart;
}
```

Now it is possible to write algorithms which will sort this and any other class that implements the interface. For example, an array sort:

```
package interfaceExamples;
public class ArraySort {
  public static void sort (Ordered[] oa) {
    Ordered tmp;
    int pos;
    for (int i = 0; i < oa.length - 1;  i++) {
      pos = i;
      for (int j = i + 1; j < oa.length; j++) {
        if (oa[j].lessThan(oa[pos])) {
          pos = j;
        }
      }
      tmp = oa[pos];
      oa[pos] = oa[i];
      oa[i] = tmp;
    }
  }
}
```

The `sort` method takes two arguments; the first is an array of objects that implement the `Ordered` interface and the second is the number of items in the array. The implementation performs an exchange sort. The important point about this example is that when two objects are exchanged, it is the reference values which are exchanged and hence it does not matter what type the object is (as long as it supports the `Ordered` interface).

To use the above classes and interfaces simply requires the following

```
{
  ComplexNumber[] arrayComplex = { // for example
                  new ComplexNumber(6f,1f),
                  new ComplexNumber(1f, 1f),
                  new ComplexNumber(3f,1f),
                  new ComplexNumber(1f, 0f),
                  new ComplexNumber(7f,1f),
                  new ComplexNumber(1f, 8f),
                  new ComplexNumber(10f,1f),
                  new ComplexNumber(1f, 7f)
  };
  // array unsorted
  ArraySort.sort(arrayComplex);
  // array sorted
}
```

In fact, the package `java.lang` already defines an interface called `Comparable` with a function called `compareTo` which could be used in place of `Ordered`. Furthermore, there is a static method in `java.util.Arrays` called `sort` that implements a merge sort of objects.

Interfaces have three further properties which should be noted. Firstly, like classes, they can participate in inheritance relationships with other interfaces. However, unlike classes, multiple inheritance is allowed. Secondly, a class can implement more than one interface, and hence, much of the functionality of multiple inheritance can be achieved for classes. Finally, interfaces also provide the mechanisms by which callbacks can be implemented. This is when an object (server) needs to call one or more methods defined in the caller's (client’s) class. The client’s class can be of any type. As Java does not support function pointers, it is necessary to provide an alternative mechanism to that used traditionally in languages like C and C++. If the class implements an interface which defines the required functions, then the client can pass itself as a parameter to the server. The server can then call-back on any of the methods defined in that interface.

10 Inner classes

Inner classes are classes that are declared within other classes, they allow more control over the visibility and give added flexibility. There are various types of inner classes, the most important for this book are **member**, **local** and **anonymous** classes.

Member classes

A member class is declared in the same way that a method or a field is declared. The code of the member class can access all fields and methods of the enclosing class.

```
public class Outer {
  // local fields and methods of Outer
  class MemberClass {
    // Fields and methods of the Member class can
    // access local fields and methods of the
    // Outer class.
  }
}
```

Local classes

A local class is declared within a block of code within an enclosing class (such as within a method of the enclosing class). It can access everything that a member class can access. It can also access any final variables declared locally to the block declaring the local class (and final parameters, if the block is a method).

```
public class Outer {
  // local fields and methods of Outer
  void method(/* parameters */) {
    // local variables
    class LocalClass {
      // Fields and methods of the LocalClass
      // can access local fields and methods of
      // the Outer class, and final fields or
      // parameters of the declaring method.
    }
  }
}
```

Anonymous classes

An anonymous class is a local class definition that has no name and is used to immediately create an instance of that class. It is often employed to create a class that implements an interface without having to give a name to the class definition.

```
public class Outer
  // local fields and methods on Inner

  Ordered method(/* parameters */) {
    // local variables
    return new Ordered()
      {
        // Here is the code for the anonymous class
        // that implements the Ordered interface.

        // Its fields and methods can access local fields
        // and methods of the Outer class, and final
        // fields or parameters of the declaring method.
      }
   }
}
```

11 Summary

This document has provided some necessary introductory material on Java for those not familiar with the language but have experience in C, C++ and are aware of the basic principles of object-oriented programming.for the remainder of the book. The book assume that the reader is familiar with sequential programming in Java.