



Basic Types

boolean	logic	1 bit	true false	
byte	8-bit signed integer	1 bytes	-128 to 127	
short	16-bit signed integer	2 bytes	-32768 to 32767	
int	32-bit signed integer	4 bytes	-2147483648 to 2147483647	
long	64-bit signed integer	8 bytes	-2^{63} to $2^{63} - 1$	
char	16-bit unsigned integer, presenting Unicode chars	2 bytes	0	to 65535
float	Single precision IEEE 754 floating point	4 bytes	1.4^{-45}	to 3.4^{38}
double	Double precision	8 bytes	4.9^{-324} to 1.8^{308}	

Identifiers in Java can contain any combination of upper-case, lowercase, numbers, ‘_’ and ‘\$’ but must not begin with a number.

Assignments

```
++i // increment, deliver new value
i++ // increment, deliver old value
--i // decrement, deliver new value
i-- // decrement, deliver new value
```

Assignment can be combined with other operators:

```
i += 4 // plus-and-becomes ie i = i + 4
i -= j // minus-and-becomes ie i = i - j
f *= 4.2 // multiply-and-becomes ie f = f * 4.2
a &= b // logical &-and-becomes ie a = a & b
i += 1 // normally you would see i++
```

Data type conversion through wrapper classes (more on this later):

```
String number = "123";
int i = Integer.valueOf(number).intValue();
```

Loops

```
while (condition) {
    statements;
}
```

The "while" loops above performed the test first, and then executed the loop. Sometimes you may wish to test at the end of the loop, after the execution of the statements in the body of the loop (and hence to execute the loop body always at least once):

```
do {
    statements;
} while (condition);
```

```
for (initialization; condition; increment) {
    statements;
}
```

in any loop, ‘break;’ causes the loop to be abandoned. ‘continue;’ means skip to the end of the statement block.

Conditionals

```
if (condition) {  
    statements;  
}  
else {  
    statements;  
}
```

(else branch not compulsory)

```
switch (expression) {  
    case value1: statements  
                statements  
    case value2: statements  
    default:     statements (optional)  
}
```

N.B. Include 'break;' statements after each case, otherwise all cases will execute one after the other.

Arrays

May be declared *either* as datatype[] identifier *or* datatype identifier[].

After declaration, must allocate memory: identifier = new datatype [length];

You can initialise arrays in one line: int[] myArray = {1,3,5,7,9};

Multidimensional arrays:

```
int [ ] [ ] table = new int [width] [height]
```

```
or int[ ][ ] table = { { 1, 2, 3, 4, 5 }, { 5, 4, 3, 2, 1 }, { 1, 3, 5, 3, 1 } };
```

Java stores multidimensional arrays as arrays of arrays, so they need not be rectangular.

Bubble Sort

```
for (int j=0; j<charArray.length; j++) {  
    for (int i=0; i<charArray.length-1; i++) {  
        if (charArray[i] > charArray[i+1]) {  
            char tempCh = charArray[i];  
            charArray[i] = charArray[i+1];  
            charArray[i+1] = tempCh;  
        }  
    }  
}
```

Classes, Fields and Methods

If a local identifier clashes with a class constant or variable, or with the name of another method, then that global variable/method becomes inaccessible within this method. The local variable/method will be referred to by the identifier.

Classes can have multiple constructors if they have different number of arguments, or different type of arguments.

The current instance of the class can be called using 'this'.

Good programming style to declare member variables as private (see later) and get/set them using getter and setter methods in the class (which are accessible).



Access Modifiers

private – accessible only from within the same class
default – accessible only from within the same package
protected - accessible only from within the same package and subclasses
public – accessible anywhere

Static and Final Modifiers

static – only created once. All instances share the one field. (variables). We can call a static method/variable without having to create an instance of the class containing it.

final – cannot be subclasses (classes); cannot be modified (variables).

Basic vs Reference Types

When a storage location is created for a variable of a basic type (int, char,...) it actually holds the **value** of the variable. When an object is created, also storage locations for its fields are created. An object itself is stored as a **reference** to the group of those locations.

Therefore, two objects are equal (==) only if they occupy the same memory location. Objects have an equals(Object o) method that can be overridden to compare each field individually. Similarly, if you assign one object to another, it simply points to the same location in memory. The clone() method can be overridden to create new memory for each field.

N.B. Arrays are passed as objects.

N.B. In constructors, it is often a good idea to make *deep* copies of any reference type variables, i.e. allocate it different memory space.

Extending Classes, Inheritance & Polymorphism

```
public class Child extends Parent
```

Child inherits fields and methods from Parent. Can be overridden in the child class.

In the context of Child, Parent is referred to as super. Hence parent's methods are super.method() and parent's constructor is super().

We can use Child where Parent is expected (Child is just a special case of Parent) but not the other way around. We can even declare a Child object as type Parent: Parent c = new Child().

N.B. In this case fields/methods of c can only be called if Parent has them too, however, when called, the Child version of the method/variable will be used. This is because the compiler checks the declared type, whereas the run-time system checks the actual type.

All objects extend Object.

Object has the following methods:

Object clone() throws CloneNotSupportedException - creates and returns a copy of this object;

boolean equals(Object o) - indicates whether some other object is equal to this one.

Class getClass() - returns the runtime class of an object.

int hashCode() - returns a hash code value for the object. (unique number for this object)

String toString() - returns a string representation of the object. (name+@+hexadecimal hashCode)

These methods are often overridden.

Abstractions & Class Hierarchies

Sometimes when defining a class at the high level of generality one wants to guarantee that a certain method exists but cannot provide implementation for this method which would work for all classes extending the given one.

A method can be declared as *abstract* without providing implementation.

- A class which contains abstract methods must be declared *abstract*.
- It is not possible to create instances of an abstract class.
- A class which does not contain abstract methods can also be declared abstract. This is done to make it impossible to create instances of a class.
- Abstract classes are used to keep the relevant code together at the right place in the class hierarchy, and make it easier to define subclasses.
- Every non-abstract class which extends an abstract class should provide implementation for all abstract methods.

Interfaces

- An *interface* is a type which contains only abstract methods and related constants, classes and interfaces.
- Any class which *implements* an interface is guaranteed to provide its methods.
- Interfaces provide only design whereas classes provide both design and implementation.
- Instances of a class which implements an interface I can be treated as being of type I.
- A class can implement several different interfaces (public class implements interface1, interface 2 {})

Interfaces are types - an object can be declared to be of type I, where I is an interface. However, interfaces cannot be instantiated, and they are more abstract than abstract classes.

Interface methods are always abstract (no implementation), so the *abstract* keyword is omitted. Methods are always public. Methods are never static, because static methods are class-specific. Fields are always static and final (constants used to define methods).

Exceptions

Exceptions are a clean way to check for errors without cluttering code. They also provide a mechanism to signal errors directly and handle them. An exception is *thrown* when an error condition is encountered.

If an exception is thrown, there are two possible scenarios:

- the exception is caught in a try, catch block and handled
- the exception is not caught by the method which was active when the exception was thrown. It is passed to the method which called it, and so on, until it is caught by the default exception handler. Default exception handler terminates the program and prints some information about the exception and where it was thrown.

Exception() - constructs an Exception with no specified detail message.

Exception(String s) - constructs an Exception with the specified detail message.

Exceptions which a method throws are as important as its return type.

If a method throws a checked exception and does not catch it, it should be declared with a *throws* clause.



Casting, ADTs, Collections, Wrapper classes

As already discussed, basic types are stored and passed by value whereas reference types (objects) are stored and passed by reference. *null* is not of any type, not even Object. It can be used anywhere a reference is expected.

Casting

Casting is explicitly converting from one data type to another data type.

For example, we can cast an int as a double or a double as an int (losing information in the latter case). We can also cast one Object type as another, provided it

Abstract Data Types

Data type: data + operations (e.g. concrete class in Java)

Abstract data type (ADT) - tells which data it holds and what are the operations it can perform but no implementation details (e.g. an interface in Java).

The Collection Interface - public abstract interface Collection

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

- boolean add(Object x) add an element to the collection
- boolean contains(Object o) Returns true if this collection contains the specified element.
- boolean isEmpty() Returns true if this collection contains no elements.
- int size() Returns the number of elements in this collection.

The List ADT

A *list* is a variable length sequence of objects which supports the inserting and removing elements and traversing the list. Below are some methods of the Java List Interface (which extends Collection).

- boolean add(Object x) add an element x to the end of the list;
- void add(int i, Object x) add an element at position i in the list;
- Object remove(int i) remove the element at position i in the list;
- Object get(int i) returns the element at position i in the list;
- int size() returns the number of elements in the list;

Lists are used when you do not know in advance how many objects you need to store.

Some Lists:

- LinkedList
- ArrayList
- Vector

These three can be constructed using an existing Collection, i.e. list = new LinkedList (Collection c)

- Hashtable – contains data as pairs of keys and values.

Summary

- Reference types can be converted to each other. To promote an object up the class hierarchy (treat it as a more general type, e.g. treat a Safe as an Object) casting is not needed. The opposite direction requires explicit casting.
- Casting succeeds only if the actual type of the object which is cast is the same or more specific than the type it is cast to.
- To use a basic type where an Object is expected, put the basic type in a wrapper.
- An abstract data type is a description of a data type which says which data it holds and which operations can be performed on the data, but does not give any implementation details.



Threads

A thread is a sequence of steps executed one at a time.

Race hazard

A race hazard occurs when several threads are racing each other trying to access the same shared resource, and may modify it in an interleaved way.

Cure: Synchronisation – the threads should be declared synchronised to avoid interleaving – when a thread accesses a variable, it ‘locks’ it from any other threads.

A thread, therefore, should be able to i) lock an object, ii) wait for a resource and iii) notify other threads when the resource is available.

Fairness, Starvation and Deadlock

- A system is *fair* when each thread gets enough access to limited resource to make reasonable progress.
- *Starvation* occurs when one or more threads are blocked from gaining access to a resource and thus cannot make progress.
- *Deadlock* occurs when two or more threads are waiting on a condition that cannot be satisfied. For example, Thread 1 is waiting for Thread 2 to give it resource A and Thread 2 is waiting for Thread 1 to release resource B.

Implementation

Either:

- subclass the Thread class and override its run() method (the default implementation does nothing).
- implement Runnable interface (has the run() method).

Some Thread methods

- Constructors: Thread(); Thread(Runnable t); ...
- void run()
- void start()
- static void sleep (long millis) throws InterruptedException
- void setPriority (int newPriority): threads may have different priority and scheduled accordingly.
- wait(), wait(long timeout), notify(), notifyAll() inherited from Object
- yield() - give another thread a chance

The stop() method is deprecated – don't use it. Instead use ‘return;’

Synchronized Methods

Blocks of code and methods which access the same object from separate threads are called critical sections. They are identified with synchronized keyword: public synchronized double getBalance()

The thread which called a synchronised method gets a lock on the object whose method was called. Other threads cannot call a synchronised method on the same object until the object is unlocked.

Tell other threads if you changed something they may be interested in:

```
synchronized void changeCondition() {  
    ... change some value used in a condition test  
    notifyAll();  
}
```

Threads are subprocesses running within the same program. They can compete for resources; the programmer should ensure fairness. Threads can be synchronised, so that one thread gets access to an object only when another is finished. They can wait for one another and notify each other of changes.



I/O & Networking

Writing to a File

```
FileWriter fw = new FileWriter(fileName);
fw.write(string);
fw.close();
```

Reading to a File

```
FileReader fr = new FileReader(fileName);
String fileContents = new String();
int c = fr.read(); // -1 is end of file character
while(c != -1) {
    fileContents = fileContents + (char) c;
    c = fr.read();
}
// close the file, otherwise waste memory
fr.close();
return fileContents;
```

or, to read a whole line at a time, use `BufferedReader`:

```
BufferedReader in;
String s;

try {
    in = new BufferedReader(new FileReader(filename));
    while ((s = in.readLine()) != null) {
        System.out.println("Echo line:" + s);
    }
}
catch (Exception e) {
    ...
}
```

Networks

URLS

The simplest constructor (takes a String): `URL school = new URL("http://www.cs.nott.ac.uk/");`
It throws a `MalformedURLException`.

Methods:

- `boolean equals(Object obj)` - compares two URLs.
- `String getFile()` - returns the file name of this URL.
- `String getHost()` - returns the host name of this URL, if applicable.
- `int getPort()` - returns the port number of this URL.
- `InputStream openStream()` - opens a connection to this URL and returns an `InputStream` for reading
- from that connection.
- `URLConnection openConnection()` - returns a `URLConnection` object that represents a connection to the remote object referred to by the URL.

Reading from a URL

```
BufferedReader in = new BufferedReader (new InputStreamReader ( example.openStream() ) );
```



Ports and Sockets

URLs are a high-level mechanism for accessing resources on the Internet.

Client-server applications require lower-level network communication.

A *port* is an abstraction of a physical place through which communication can proceed between a server and a client. A *socket* is an abstraction of a network software which enables communication in an out of the program. Several sockets (for connecting clients) can be created on a single port (server).

ServerSocket

One of the constructors:

ServerSocket (int port) - creates a server socket on a specified port.

Some methods:

- Socket accept() - listens for a connection to be made to this socket and accepts it.
- void close() - closes this socket.
- InetAddress getInetAddress() - returns the local address of this server socket.
- int getLocalPort() - returns the port on which this socket is listening.

(Client) Socket

Some of the constructors:

- Socket(String host, int port) - - creates a stream socket and connects it to the specified port number on the named host.
- Socket(InetAddress address, int port) - creates a stream socket and connects it to the specified port number at the specified IP address.

Some methods:

- void close() - closes this socket. InetAddress getInetAddress() - returns the address to which the socket is connected.
- int getLocalPort() - returns the local port to which this socket is bound.
- OutputStream getOutputStream() - returns an output stream for this socket.
- InputStream getInputStream() - returns an input stream for this socket.

Implementation of a client/server application

- Write a server class (what does the server do; at least should open a ServerSocket)
 - Write a client class
 - Write a protocol for communication between client and server
 - When the server is running, it creates a thread to deal with each new client
-