Faculty of Engineering and Information Sciences
University of Wollongong

ECTE331 - Summary

Anton Kullberg

2017–10–30

Contents

Introduction **1**

1 Encryption **2**
1.1 DES . . . 2
1.2 Simplified DES . . . 3
1.3 Hash functions . . . 3
1.3.1 MD5 Secure Hash Function . . . 3
1.4 SNMP . . . 3

2 Java **4**
2.1 Principal elements of Java . . . 4

3 Real-time systems **5**
3.1 Characteristics of a real-time system . . . 6
3.2 Real-time programming languages . . . 7
3.3 Real-time systems reference model . . . 7
3.3.1 System resources . . . 7
3.3.2 Jobs and tasks . . . 8
3.4 Reliability and fault tolerance . . . 9
3.4.1 Faults . . . 10
3.4.2 Hardware fault tolerance . . . 11
3.4.3 Software fault tolerance . . . 11
3.5 Exceptions . . . 12
3.5.1 Exceptions in Java . . . 14

4 Concurrent programming **14**
4.1 Why do we need concurrent programming? . . . 14
4.2 Processes and Threads . . . 15
4.2.1 Nested tasks . . . 15
4.2.2 Thread Interference . . . 15
4.2.3 Memory Consistency Errors . . . 16
4.3 Concurrent programming in Java . . . 16
4.3.1 Thread states . . . 17
4.3.2 Synchronisation . . . 18
4.3.3 High level concurrency objects . . . 19

5 Real-time specification for Java (RTSJ **19**
5.1 Memory management . . . 21
5.1.1 Memory areas . . . 21
5.1.2 Nested memory areas . . . 22
5.1.3 Heap memory . . . 22
5.1.4 Immortal memory . . . 23
5.1.5 Scoped memory . . . 23
5.1.6 Physical and raw memory . . . 23
5.2 Time values and clocks . . . 24
5.3 Schedulable objects and schedulers . . . 25
5.3.1 Schedulable object parameters . . . 26
5.3.2 The Schedulable interface . . . 27

5.3.3 Schedulers . . . 27
5.3.4 Real-time threads . . . 28
5.4 Scheduling . . . 28
5.4.1 Fixed priority scheduling (FPS) . . . 29
5.4.2 Schedulability analysis . . . 30
5.4.3 Simple process model . . . 31
5.4.4 Asynchronous events and event handlers . . . 32
5.4.5 Asynchronous transfer of control . . . 33
5.5 Synchronisation . . . 34
5.5.1 Priority inversion . . . 35

6 Java on the web **35**
6.1 Web applications . . . 35
6.2 Servlets and JSP . . . 36
6.2.1 Creating and mapping Servlets . . . 37
6.2.2 Getting parameter values into servlets . . . 38
6.2.3 Get and set request object attributes . . . 38
6.2.4 Forwarding a request object . . . 38
6.2.5 Validating data on the server . . . 39
6.2.6 Initialisation of parameters and servlets . . . 39
6.2.7 Servlet methods . . . 39
6.2.8 JavaBeans . . . 40
6.2.9 Including files in JSP . . . 41
6.2.10 JSP tags . . . 42
6.3 Java Applets . . . 42

Introduction

This document is a summary of the course ECTE331 - Real-time Embedded Systems at the University of Wollongong during spring session 2017. The summary is based on the lectures of Dr. Peter Vial. It is not in the same chronological learning order as in the course itself but is instead divided up into ”natural” sections where each section treats some part of the course.

1 Encryption

Conventional encryption has five ingredients:

1. **Plaintext** - human readable messages
2. **Encryption algorithm** - substitutions and transformations on the plaintext
3. **Secret key** - An input to encryption/decryption which is needed for these
4. **Ciphertext** - Scrambled output message
5. **Decryption algorithm** - Reverse process of encryption

There are two distinct approaches to crack an encryption:

- **Cryptanalysis** - Relies on the nature of the algorithm and characteristics of the plaintext or some plaintext-ciphers
- **Brute force attack** - On average half of the possible keys need to be tested to achieve success

See table 1 for the time it takes to crack an encryption of different key lengths.

Table 1: Time to break encryption at different key lengths

Key size (bits)	**Number of alternative keys**	**Time required at 1 key/μs**	**Time required at 10^6 keys/μs**
32	$2^{32} = 4.3 * 10^9$	$2^{31}\mu$s = 35.8 min	2.15 ms
56	$2^{56} = 7.2 * 10^{16}$	$2^{55}\mu$s = 1142 years	10.01 hours
128	$2^{128} = 3.4 * 10^{38}$	$2^{127}\mu$s = $5.4 * 10^{24}$ years	$5.4 * 10^{18}$ years

1.1 DES

DES uses a Feistel Cipher Structure. The feistel structure is a particular form of a product cipher proposed by Claude Shannon in 1948. A product cipher is where two or more basic ciphers (permutation and substitution) are done in series such that the final product is cryptographically stronger than any of its component ciphers. Virtually all current symmetric block ciphers use or are based on the Feistel structure. Attributes of DES:

- Uses 64-bit blocks of text to encrypt
- Uses a 56-bit key
- Produces a 64-bit output ciphertext
- Uses 16 rounds of permutations/substitutions

DES can operate in different modes. The simplest mode is ECB (Electronic Code Book) mode. In ECB, plaintext is handled 64-bits at a time and each block is encrypted using the same key. This means that if any two 64-bit blocks of plaintext are the same, they will produce the same ciphertext. This could potentially be a target for cryptanalysis if the message is highly structured. A simple way to get around this is to use the CBC (Cipher Block Chaining) mode. Here the input to the encryption is the current plaintext block X-ORed with the ciphertext output of the previous 64-bit block. This ensures that two identical plaintext blocks do not produce the same ciphertext.

1.2 Simplified DES

Simplified DES is a simplified version of the full DES. It was developed to help with understanding of DES. It has similiar properties to full DES but uses a 10-bit encryption key and hence is not very secure. The simplified DES algorithm is based on the Feistel Cipher Structure. See figure 1 for a visual explanation.

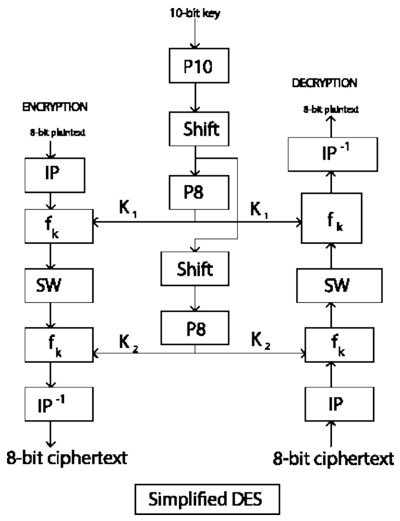


Figure 1: Simplified DES

The difference between SDES and DES are depicted in table 2.

Table 2: DES compared to SDES

	SDES	DES
Input/output length	8-bit	64-bit
Key length	10-bit	56-bit
F rounds	2	16
S-boxes	2	8

1.3 Hash functions

An essential element of most authentication and digital signal schemes is a secure hash function. A hash function accepts a variable length message as input and produces a fixed size hash code which can also be called the message digest. The message digest can then be attached to message to authenticate it.

1.3.1 MD5 Secure Hash Function

The MD5 Secure Hash Function is specified in RFC1321. It was developed by Ron Rivest at MIT. It takes a message of arbitrary length and produces a 128-bit message digest. Input is processed in 512-bit blocks.

1.4 SNMP

SNMP is short for Simple Network Management Protocol. It is used to manage networks at the application layer using datagrams (UDP packets). SNMPv3 is the current version which has added security, encryption, timeliness and hashing to make it more secure. SNMPv3 specifies the use of four different cryptographic algorithms:

1. Conventional encryption with DES (Data Encryption Standard)
   - Symmetric encryption or single key encryption
   - Uses plaintext and requires a special coding key or an alternate alphabet
2. MD5 Secure Hash Function
   - A hash function accepts a variable length message M as input and produces a fixed size hash code as output
   - MD5 is processed in 512-bit blocks
3. SHA-1 Secure Hash Function
   - Processed in 512-bit blocks
   - Outputs a 160 bit message digest
4. Message Authentication with HMAC
   - HMAC (Hash Message Authentication Scheme)
   - RFC 2104 provides design details of HMAC
   - Uses a secret key
   - Generates a small block of data known as the Message Authentication Code (MAC)

2 Java

Java is an object-oriented programming language like C and C++. It is platform-independent which means that a program constructed with Java will run on a PC, MAC or UNIX workstation. Device-independent graphics are also built directly into the language. For more information about Java, check the Java specification.

2.1 Principal elements of Java

- The Java programming language
- The Java Virtual Machine (JVM)
  - A ”virtual” computer that executes Java code identically regardless of the host computer
  - There must be a custom JVM for each type of computer executing Java programs
- The Java Application Programming Interface (API) - Defines the standard classes built into the language

Packages built into Java can be seen in table 3.

Table 3: Java packages

Import statement	Title	Description
java.applet	The Java Applet Package	Contains all of the classes and interfaces required to support Java Applets
java.awt	The Abstract Windowing Toolkit Package	Contains many of the classes needed to support old-style graphical user interfaces. Parts of this package are also used with the new "Swing" GUI package
java.beans	The Java Beans Package	Contains classes that enable programmers to create reusable software components
java.io	The Java Input/Output Package	Contains classes that allow a programmer to input/output data
java.lang	The Java Language Package	Contains basic classes and interfaces required by most Java programmers
java.net	The Java Networking Package	Contains classes that allow a program to communicate via networks
javax.swing	The Swing Package	Contains many of the classes and interfaces required to support the new "Swing" GUI
java.text	The Java Text Package	Contains classes and interfaces that allow a program to manipulate numbers, dates, characters and strings
java.util	The Java Utility Package	Contains utility classes and interfaces; data and time manipulations, random number generation etc.

3 Real-time systems

A system is a set of components that interact to accomplish an objective. A system is a mapping of a set of inputs to a set of outputs. Every real-world entity, either synthetic or natural, can be modelled as a system. A real-time system is a system with explicit timing requirements:

- The system must respond to external inputs within a specified period
- Failure to meet timing requirements is as bad as producing the wrong outputs

Examples of timing requirements:

- The system must start a job every T seconds
- The system must complete a job no later than D seconds after it starts

A real-time system does not necessarily have to process data quickly but it must have explicitly defined response times. Real-time computing is concerned with designing systems with a guaranteed timing performance. There are three distinct classifications of a real-time system:

1. **Hard real-time:**
    - Failure to meet a single deadline leads to complete and catastrophic system failure
    - Recovery may be difficult or futile
    - Examples: Safety critical systems
2. **Soft real-time:**

- Performance is degraded but not destroyed by failure to meet the timing constraints
- Late results are still useful to some degree

3. **Firm real-time:**
    - A category sometimes seen in literature
    - Considered as an intermediate between hard and soft real-time

Classification examples can be seen in table 4.

Table 4: Examples of real-time classifications

Application	Type	Explanation
Avionics weapon delivery system	Hard	Missing launch deadlines means missing target
Embedded navigation controller for autonomous weed killer	Firm	Missing critical navigation deadlines causes the robot to veer out of control and damage crops
Automated teller machine	Soft	Missing even many deadlines will not lead to catastrophic failure, only degraded performance

There are also different types of systems:

- **Time-aware** - System makes explicit reference to time (e.g. open vault door at 09:00)
- **Reactive** - System must produce output within deadline (as measured from input)
    - Control systems are reactive
    - Required to constraint input and output variability, input jitter and output jitter control
- **Time-triggered** - Computation is triggered by the passage of time
    - Release activity at 09:00
    - Release activity every 25 ms
- **Event-triggered** - Computation is triggered by an external or internal event
    - The released activity is sporadic if there is no bound on the arrival interval of the event
    - The released activity is aperiodic if there is no such bound

3.1 Characteristics of a real-time system

- **Guaranteed response times** - need to be able to predict the worst case response times for the system.
- **Concurrent control of separate system components** - devices operate in parallel in the real-world, model this by concurrent entities in the program
- **Facilities to interact with special purpose hardware** - need to be able to program devices in a reliable and abstract way
- **Support for numerical computation** - be able to support the discrete/continuous computation necessary for control system feed-back and feed-forward algorithms

- **Large and complex -** vary from a few hundred lines of assembler or C to 20 million lines of Ada, also variety as well as size is an issue
- **Extreme reliability and safety -** embedded systems typically control the environment in which they operate; failure to control can result in loss of life, damage to environment or economic loss

3.2 Real-time programming languages

- Assembly languages
- Sequential programs implementation languages - RTL/2, Coral 66, Jovial, C
- High-level concurrent languages - Ada, Chill, Modula-2, Mesa, Java

3.3 Real-time systems reference model

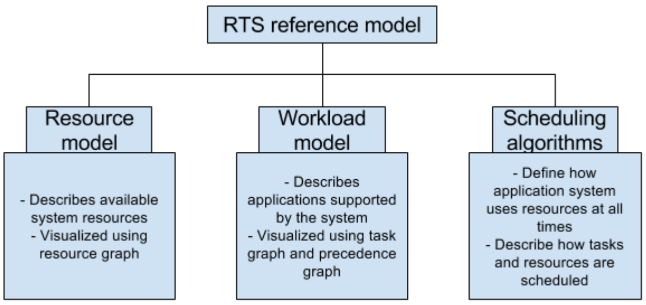

RTS reference model
Resource model
- Describes available system resources
- Visualized using resource graph
Workload model
- Describes applications supported by the system
- Visualized using task graph and precedence graph
Scheduling algorithms
- Define how application system uses resources at all times
- Describe how tasks and resources are scheduled

Figure 2: Real-time system reference model

The reference model is a generic reference model to describe the system to allow analysis and simulate its real-time performance.

3.3.1 System resources

- **Processors**
  - Carry out machine instructions
  - Move data, read/write files
  - Speed is an important attribute
  - Examples: computers, database servers, transmission links etc.
- **Resources**
  - Known as passive resources
  - Usually reusable
  - Examples: memory, sequence numbers, semaphores, mutex etc.

A job's execution time is assumed to depend on the processor's speed, not the resource's speed.

3.3.2 Jobs and tasks

- **Job** - a unit of work to be carried out by the system. Denoted J_i. Instance that occurs at run-time.
- **Task** - a set of jobs that execute to support a function of the system. Denoted T_i. Tasks are specified at design time.

A job can have several parameters. The temporal (time) parameters of a job are explained in table 5

Table 5: Temporal parameters of a job

Parameters	Symbol	Explanation
Release time	r_i	Time at which the job is ready for execution
Absolute deadline	d_i	Time by which the job must be complete
Relative deadline	D_i	Maximum allowable response time D_i=d_i-r_i
Execution time	e_i	Time required to complete the job on a single processor with no interruptions. e_i often means the maximum execution time
Feasible interval	$(r_i, d_i]$	Time interval during which the job can execute

The functional parameters of a job are explained in table 6.

Table 6: Functional parameters of a job

Parameters	Explanation
Preemptivity	Can the job be suspended at any time to allow the execution of other jobs?
Laxity	Is the job's deadline soft or hard?
Criticality	How important is the job with respect to other jobs?

Jobs also have interconnection parameters. This is to describe inter-job dependency. Jobs can have the following interconnections:

- **Independent** - Jobs that can be executed in any order
- **Dependent** - Jobs that must be executed in a particular order
- **Predecessor** - If job J_i must complete before J_k can begin, then J_i is a predecessor to J_k
- **Immediate predecessor** - If J_i is a predecessor to J_k and no other jobs are executed between them, then J_i is an immediate predecessor to J_k

Job dependency can be described with a precedence graph. Each point in the graph represents one job and a directed edge represents a job relationship. In figure 3, task A is a predecessor to task E and D as well as an immediate predecessor to task B. It is also independent of task C.

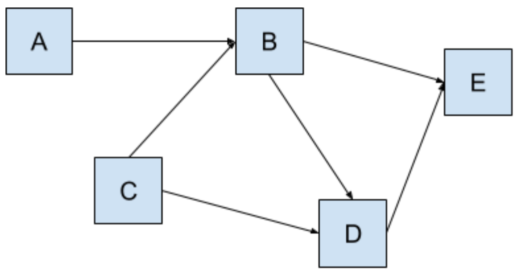


Figure 3: Precedence graph

An extension of a precedence graph is a so called task graph. In a task graph a jobs feasible interval is also included. Sometimes job duration is also included as well as how many of a job's immediate predecessors need to complete before the job is executed.

3.4 Reliability and fault tolerance

The dependability of a system is of utmost importance. Dependability is often divided up in the following:

- **Attributes:**
  - Availability
  - Reliability
  - Safety
  - Confidentiality
  - Integrity
  - Maintainability
- **Means:**
  - Fault prevention
  - Fault tolerance
  - Fault removal
  - Fault forecasting
- **Impairments:**
  - Faults
  - Errors
  - Failures

The reliability of a system is the measure of the success with which it conforms to some authoritative specification of its behaviour. When the behaviour of a system deviates from that which it is specified for it's called a failure. Failures result from unexpected problems internal to the system and eventually manifest themselves in the system's external behaviour. These problems are called errors and their mechanical or algorithmic cause are termed faults. These often have a circular dependence as such: failure - fault - error - failure - fault.

3.4.1 Faults

There are different types of faults:

- **Transient fault** - Starts at a particular time, remains in the system for some time and then disappears.
- **Permanent fault** - Remain in the system until they are repaired.
- **Intermittent fault** - Transient fault that occurs from time to time.
- **Software fault** - Called "bugs" -
  - Bohrbugs - reproducible and identifiable
  - Heisenbugs - only occur under rare conditions, hard to reproduce and identify

  Software doesn't deteriorate with age, but faults can lay dormant for long periods - such as memory leaks.

There are different ways to treat faults:

- **Fault prevention** - Attempts to eliminate any possibility of faults before the system goes operational
- **Fault tolerance** - Enables a system to continue functioning even in the presence of faults

Fault prevention is done in two stages, fault avoidance and fault removal. Fault avoidance attempts to limit the introduction of faults during system construction by:

- Use reliable components
- Use of thoroughly refined techniques
- Package the hardware to screen out expected forms of interference
- Rigorous specification of requirements
- Use of proven design methodologies
- Use of languages with facilities for data abstraction and modularity
- Use of software engineering environments to help manipulate software components and thereby manage complexity

Fault removal is basically the procedure for finding and removing the cause of errors. System testing can never be exhaustive and remove all potential faults so they will always exist. Basically, a test can only show the presence of faults, not their absence. An alternative to fault prevention is fault tolerance.

A system can be fault tolerant in different levels:

- **Full fault tolerance** - The system continues to operate in the presence of faults, with no significant loss of functionality or performance.
- **Graceful degradation** - The system continues to operate in the presence of faults, accepting a partial degradation of functionality or performance during recovery or repair.
- **Fail safe** - The system maintains its integrity while accepting a temporary halt in its operation.

The level required will depend on the application. Most safety critical systems require full fault tolerance but will in practice often settle for graceful degradation.

3.4.2 Hardware fault tolerance

All fault tolerance levels rely on extra elements introduced in the system to detect and recover from faults. This is called redundancy. The goal should be to minimise redundancy while maximising/maintaining reliability. There are two types of hardware redundancy - static and dynamic.

- **Static hardware fault tolerance** - Redundant components are used inside a system to hide the effects of faults.
- **Dynamic hardware fault tolerance** - Redundancy supplied within a component which indicates that the output is in error. Recovery must be provided by another component.

3.4.3 Software fault tolerance

Software redundancy/fault tolerance can also be static and dynamic but manifest themselves a little bit differently:

- **Static software fault tolerance** - N-version programming.
- **Dynamic software fault tolerance** - Detection and recovery. Recovery blocks: backward error recovery. Exceptions: forward error recovery.

Error detection is done in one of two environments, environmental or application:

- **Environmental detection**
  - Hardware - e.g. illegal instruction
  - O.S/RTS - null pointer
- **Application detection**
  - Replication checks
  - Timing checks
  - Reversal checks
  - Coding checks
  - Reasonableness checks (e.g. assertion)
  - Structural checks (e.g. redundant pointers in linked list)
  - Dynamic reasonableness checks

Error recovery has two distinct approaches, forward and backward:

- **Forward error recovery** - System continues from an erroneous state by making selective corrections to the system state.
  - Make sure the controlled environment is safe (may be damaged due to failure)
  - System specific. Depends on accurate predictions of the location and cause of the error.
  - Examples: redundant pointers in data structures and the use of self-correcting codes such as Hamming codes.
- **Backward error recovery** - Restoring the system to a previous safe state and execute an alternative section of the program.

  - Same functionality but uses a different algorithm (N-version programming).
  - The state to which the system is restored is called a recovery point
  - Advantage: erroneous state is cleared and does not rely on finding the cause of the fault
  - Disadvantage: cannot undo errors in the environment

The last type of software fault tolerance is N-version programming. This is the generation of functionally equivalent programs from the same initial specification. The different versions are run concurrently and results from each one is counted as a ”vote”. The result with the most votes is considered to be correct. N-version programming relies on:

- **Initial specification** - The majority of software faults stem from inadequate specification. This would manifest itself in each version of the program.
- **Independence of effort** - Experiments produce conflicting results. Complex parts of the specification lead to a lack of understanding of the requirements.
- **Adequate budget** - The predominant cost is software. A 3-version system will triple the budget requirements and cause problems of maintenance.

3.5 Exceptions

An exception is an event that interrupts the normal processing flow of a program.

- When a Java method cannot complete its normal processing, it is said to ”throw” an exception
- Exceptions are objects of the Exception class or of some subclass of the Exception class

There are two types of exceptions: checked exceptions and run-time exceptions.

- **Checked exceptions** - Exceptions that must be explicitly declared to the Java compiler. If a checked exception could occur within the scope of a class and it is not declared, the compiler reports an error.
- **Run-time exceptions** - Exceptions that can occur anywhere within a program, so they are not explicitly checked for by the Java compiler.

Exceptions can be either synchronous or asynchronous as well as being detected by the environment or the application. The situations are as follows:

- Detected by the environment and raised synchronously - e.g. array bounds error or divide by zero
- Detected by the application and raised synchronously - e.g. the failure of a program-defined assertion check
- Detected by the environment and raised asynchronously - e.g. an exception raised due to failure of some health monitoring mechanism
- Detected by the application and raised asynchronously - e.g. one process may recognise that an error condition has occurred which will result in another process not meeting its deadline or not terminating correctly

A synchronous exception is raised immediately as a result of a process attempting an inappropriate operation. An asynchronous exception is raised some time after the operation causing the error; it may be raised by the process executing the operation or in another process.

Every Java class must either handle (catch) any checked exception that can occur within it's scope or else explicitly throw it. Every exception thrown by a method must either be caught and handled or re-thrown. If every method above the throwing method in the calling tree simply re-throws the exception, the program will print out an error message and abort. The exception simply must be handled somewhere.

Exceptions are handled by including the statements which can throw an exception within a try/catch structure. The statements within the try clause are the statements which can throw an exception. If they throw an exception, it is ”caught” by the catch clause and the code within that clause is executed. A catch block only catches the specific exception named in it or it can be set to catch all exceptions. After a try-catch block a finally clause can be attached. This is always executed even if an exception is thrown. As of Java 7 there is a new exception handling mechanism which can automatically close resources used within a try-catch block.

```
//Try-catch without Automatic Resource Management

BufferedReader br = new BufferedReader(new FileReader(path));

try {
    return br.readLine();
} finally {
    if ( br!= null) br.close();
}

//Try-catch with Automatic Resource Management

try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
}
```

A function must specify a list of throwable checked exceptions ”throw A, B, C”. If the function tries to throw any other exception, a compilation error occurs.

If an exception is thrown and then handled either by the invoker or up the chain of execution, there are then two alternatives to what can happen. Either the invoker can continue execution after the exception is handled, or it can terminate and let some function up the execution chain keep on executing. There are three models of this:

- **Resumption/notify model** - The invoker continues executing after the exception is handled. A strict resumption model is difficult to implement as it is hard to repair errors raised by a real-time system. A compromise would be to re-execute the block causing the exception. The advantage of a resumption model is that an asynchronous exception can easily be handled since it typically has nothing to do with current execution.
- **Termination/escape model** - The invoker is terminated after the exception is handled. With the termination model, the exception is propagated until a handler is reached. This can cause many functions to terminate execution because of an exception in one sub-method. But because errors are often hard to repair, it is advantageous as one error might cause several others if left unchecked.
- **Hybrid model** - The handler chooses whether the invoker continues executing or not

Exception handling is a type of forward error recovery. Exceptions can be used to indicate that something went wrong and can that can be handled. There are a number of requirements for an exception handling facility:

1. The facility must be easy to understand

2. The code for exception handling should not obscure understanding of the program's normal execution
3. Should be designed so that run-time overheads are only incurred when handling an exception
4. The mechanisms should allow the uniform treatment of exceptions detected both by the environment and by the program
5. Should allow recovery actions to be programmed

3.5.1 Exceptions in Java

Java exceptions are all a subclass of the Throwable class. Java also defines other classes, e.g. Error, Exception and RuntimeException. Java implements the termination model for exception handling but it is integated into the object oriented model (all exceptions are subclasses). The Throwable hierarchy looks like in figure 4.

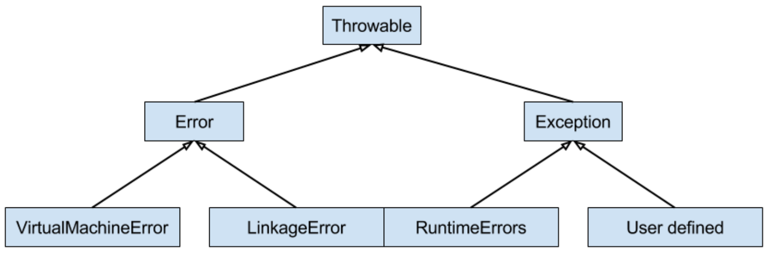


Figure 4: Throwable class hierarchy

4 Concurrent programming

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronisation and communication problems. Concurrent programming provides an abstract setting in which to study parallelism without getting bogged down by implementation details. Concurrent programming is basically the concept of allowing several things to occur at the same time (or at least give the illusion of them occurring at the same time).

4.1 Why do we need concurrent programming?

- To model parallelism from the real world
- Virtually all real-time systems are inherently concurrent - devices operate in parallel in the real world
- To allow the expression of potential parallelism so that more than one computer/processor can be used to solve a problem

The alternative to concurrent programming is sequential programming. If programmed sequentially, the programmer needs to construct a system which cyclically executes different parts of the program. The resulting program will be very complex and obscure. A concurrent program is a collection of sequential tasks executing in parallel. Each task has a single thread of control. The implementation (execution) can take three forms:

- **Multiprogramming** - Tasks multiplex their executions on a single processor
- **Multiprocessing** - Tasks multiplex their execution on a multiprocessor system where there is access to shared memory

- **Distributed Processing** - Tasks multiplex their execution on several processors which do not share memory.

All concurrent programs need some sort of control mechanism. An RTSS (run-time support system) is such a mechanism. An RTSS sits between the application software and the hardware and can take on a number of different forms:

- A software structure programmed as part of the application - C/C++ use this approach
- A standard software system linked to the program object code by the compiler - Ada/Java approach
- A hardware structure microcoded into the processor for efficiency - the aJile Java processor aJ100 is an example

4.2 Processes and Threads

All operating systems provide processes which execute in their own virtual machine (VM) to avoid interference from other processes. Recent operating systems provide mechanisms for creating "threads" within the same virtual machine. Threads are basically a "lightweight" process with it's own local memory but it does not have it's own environment variables etc. Threads have unrestricted access to it's VM so the programmer must provide protection from interference from other threads. Tasks or threads may be:

- Independent
- Cooperating
- Competing

How two threads operate in relation to each other depend on what relationship they have.

4.2.1 Nested tasks

With threads, hierarchies of tasks can be created. A thread becomes a parent as soon as it creates another thread. If the thread structure of the programming language is static, it also becomes the "guardian" of the thread. The guardian thread cannot terminate until all it's "children" have terminated. In a dynamic nested thread structure, the parent and guardian may or may not be the same thread.

4.2.2 Thread Interference

Thread interference is the process where errors are introduced when multiple threads access shared data. Interference happens when two threads try to access the same data at the same time. Interference is not a problem if threads consist only of atomic actions, that is, actions that cannot be interrupted. Example:

```java
class Counter {
   private int c = 0;

   public void increment() {
      c++;
   }

   public void decrement() {
      c--;
   }
```

```
    public int value() {
        return c;
    }
}
/* If two threads reference the same Counter object and try to increment/decrement it at the same
     time, interference happens and leads to undesirable results.*/
```

4.2.3 Memory Consistency Errors

Memory consistency errors occur when different threads have inconsistent views of what should be the same data. If we look at the example above a memory consistency error could occur if say thread A increments the counter at the same time as thread B prints it out. Then thread A's increment might not be done before thread B prints it. The way to solve both thread interference as well as memory consistency errors is called synchronisation, see section 4.3.2.

4.3 Concurrent programming in Java

Java supports threads which execute within a single JVM. Native threads map a single Java thread to an OS thread and is required to get true parallelism. Concurrency can be introduced in object oriented programming in different ways:

- Asynchronous method calls
- Early return from methods
- Futures
- Active objects

Java takes the active object approach.

Java has a predefined class **java.lang.Thread** which provides the mechanism by which threads are created. To avoid all threads having to be subclasses of the Thread class, Java also uses an interface ”Runnable”. Any class wishing to express concurrent execution must either be a subclass of the Thread class or implement the Runnable interface and provide it's own implementation of the run method.

To allow threads to communicate Java threads can read and write to shared objects protected by monitors. This is possible because every object in Java is implicitly derived from the Object class which defines a mutual exclusion lock. The lock must be acquired in order to read/write to the object in question safely. The lock is acquired by labelling a method as ”synchronized” or by encapsulating statements with the ”synchronized” keyword. A thread can also wait or notify on a single anonymous condition variable. As stated before, threads can be created in two different ways:

```
//Thread creation - Subclass
public class MotorController extends Thread {
    public MotorController() {
        //Calls the Thread class constructor
        super();
    }

    public void run() {
        /* Run method overridden */
    }
}

MotorController MC = new MotorController();
//Called to start the thread
MC.start();

//Thread creation - Interface
public class MotorController implements Runnable {
    public Motorcontroller() {}

    public void run() {
        /* Run method overridden */
    }
}

MotorController MC = new MotorController();
//Create a thread to hold the Runnable object
Thread t = new Thread(MC);
//Start the thread
t.start();
```

Java threads can typically be of two types: user threads or daemon threads. Daemon threads are typically threads who provide general services and do never terminate. As soon as all user threads are terminated, daemon threads are also terminated and the main program can terminate. Creating a daemon thread is done in the same way as creating a thread, but the "setDaemon" method of the Thread class must be called before the thread is started.

To know if a thread is still running or to wait for it's termination, one can either call the isAlive method on the target object or the join method. The calling thread then waits for the target object to terminate. A thread can also call the wait method, often used to wait for a shared object. It then halts execution until another thread has called notify on the same object.

4.3.1 Thread states

There are several states a thread can be in. Depending on which state, it is either viable for execution or not. The states are:

- **Sleeping** - If a thread calls sleep(time), the thread will halt and "sleep" until the specified time has passed.
- **Ready** - The thread is ready for execution but is not executing. If a running thread calls yield(), it indicates to the scheduler that it is ready to yield its current use of the processor.
- **Dead** - The thread is terminated. If another thread calls interrupt() on a thread object, the target thread becomes "dead". A thread also becomes dead if it has finished it's execution.
- **Waiting** - If a thread calls wait() or join() on another thread object, it is set into the waiting state and does not execute. When the target thread calls notify() or enters a dead state, the calling thread becomes ready for execution again.
- **Blocked** - The thread is blocked from execution. Can happen from I/O request or interruptions.

4.3.2 Synchronisation

Java implements two ways of synchronising: synchronised methods and synchronised statements. Both methods prevent thread interference and memory consistency errors. Both methods also rely on the fact that all classes in Java are derived from the Object class, which implements an intrinsic lock which provides exclusive access to an object. If a thread ”owns” the lock of an object, other threads trying to get access of the object lock are blocked until the lock is released. A lock is automatically acquired by using synchronised methods or synchronised statements.

- **Synchronised methods** - A whole method of an object is synchronised. The calling thread is said to hold a mutual exclusion lock, which means no other threads can execute any synchronised methods encapsulated within the same object at the same time.
- **Synchronised statements** - A statement which is synchronised. Must specify what object will provide the lock (methods must not since they are associated with an object already).

Using synchronised methods may reduce concurrency by creating unnecessary blocking if only a part of a method needs synchronisation. Synchronised statements are more powerful as they can encapsulate only that which needs synchronisation.

```java
public class Synchronised() {
    int c;

    // Synchronised method
    public synchronized void increment() {
        c++;
    }

    // Synchronised statement
    public void inc_statement() {
        synchronized(this) {
            c++;
        }
    }
}
```

Synchronisation may introduce something called thread contention. This is when two threads try to access the same resource at the same time and causes the Java run-time to execute one or more threads more slowly or halt their execution completely. These issues are called **deadlock, livelock** and **starvation**.

- **Deadlock** - A situation where two or more threads are blocked forever waiting for each other.
- **Livelock** - Two or more threads respond to deadlock with remedial actions that result in an alternative deadlock situation.
- **Starvation** - A thread is unable to gain regular access to a shared object and is unable to progress. Happens when threads are ”greedy” and make shared resources unavailable for long periods of time.

When a shared resource is unavailable a thread can choose to wait for the resource to become available. It then suspends its execution until another thread notifies it that the resource has become available. This is done by calling wait() and notify() on the shared object. These methods may only be called within synchronised methods or statements, otherwise an exception is thrown. A waiting thread can be awoken either by the notify method or by another thread interrupting it.

4.3.3 High level concurrency objects

The methods described in section 4.3.2 are very low level and are adequate for most concurrency applications. However, Java also implements some other approaches:

- **Lock objects** - Support locking idioms that simplify many concurrent applications.
- **Executors** - Defines a high-level API for launching and managing threads. Executor implementation provided by **java.lang.concurrent** provides thread pool management suitable for large-scale applications.
- **Concurrent collections** - Make it easier to manage large collections of data and can greatly reduce the need for synchronisation.
- **Atomic variables** - Minimise synchronisation need and help avoid memory consistency errors.
- **ThreadLocalRandom** - Provides efficient generation of pseudorandom numbers from multiple threads.

5 Real-time specification for Java (RTSJ

Java was not designed for real-time systems, however the advantages of using Java for real-time systems are:

- Robust language - catches many errors at compile time
- Object oriented - clean modularity, easy extensibility, design by perturbation and intrinsic information hiding
- Safe object references - cannot generate General Protection or Segmentation faults
- Concurrency - threads are first class constructs
- Easy, safe synchronisation - utilises a ”monitor” concept

The cons of using Java for real-time systems:

- Java was not built for real-time
- Requires run-time garbage collection
- Supports threads, but provides inadequate scheduling control
- No predictable event processing
- No ”safe” asynchronous transfer of control

The real-time specification for Java or RTSJ was developed to expand Java by taking into account current real-time practices. The RTSJ enhances Java in the following areas:

- Memory management
- Time values and clocks
- Schedulable objects and scheduling
- Real-time threads
- Asynchronous event handling and timers

- Asynchronous transfer of control
- Synchronisation and resource sharing
- Physical memory access

The RTSJ however only really addresses the execution of real-time Java programs on single processor systems. It provides no facilities to control, say, allocation of threads to processors.

The real time specification for Java is only real-time Java technology. It's no a universal remedy, but a more precise tool than ordinary Java. In RTSJ, there are eight major sets of interfaces defined, covering the most important issues associated with using Java in embedded systems:

1. **Memory allocation regions** - Apart from normal Java heap memory from which Java objects are allocated, the RTSJ adds two additional regions - scoped memory and immortal memory. Objects created in immortal memory can never be deleted as long as the application is running. Objects created in scoped memory will not be deleted until the last thread stops using the scoped memory region at which time the scoped memory region is emptied without requiring garbage collection.

2. **Threads** - RTSJ supports normal Java threads and defines two additional kinds. The first is the RealtimeThread characterised by scheduling and release parameters and can be used to reference any of the three memory regions. The second is the NoHeapRealtimeThread that is a RealtimeThread except it cannot reference objects in heap memory. Because it cannot reference the heap, it can preempt the garbage collector any time. Therefore, it is not affected by the garbage collector at all. Furthermore, RTSJ requires a JVM must support 28 real-time priorities above the normal 10 Java priorities.

3. **Synchronisation** - RTSJ requires that conforming JVMS must prevent priority inversion for synchronised blocks. Priority inheritance must be supported and priority ceiling emulation may also be supported. These mechanism ensure that a high priority thread will never wait longer than the longest corresponding synchronised block of a lower-priority thread.

4. **Garbager collector interface** - Under RTSJ it is possible to query the longest time the underlying garbage collector can preempt the calling thread. It is also able to force the garbage collector to run when the application wishes rather than only when the memory manager decides it is necessary. This means an application can run the garbage collector more frequently, to avoid it preempting important processes.

5. **Asynchronous events** - RTSJ provides mechanisms for creating schedulable event handlers that can be connected to asynchronous events. Asynchronous events can be triggered by something called happenings, usually known as signals in a UNIX or Linux environment. One or more happenings can trigger an event and an event can be handled by one or more event handlers. In addition, an asynchronous event handler can have scheduling and dispatch parameters similar to threads and can be prevented from using the heap if desired.

6. **Asynchronous transfer of control** - Any thread is empowered by the RTSJ to fire an AsynchronouslyInterruptedException (AIE) at a real-time thread at any time. If the AIE is enabled and the target thread is not an ATC-deferred region, the target thread will begin immediately running the appropriate AIE catch block, which will be responsible for maintaining the thread's internal consistency and taking whatever action was intended. By default, threads are ATC-deferred, so Java programs unaware of ATC will not be affected.

7. **High-resolution timers** - Timer functions are available in the RTSJ that permit delays and asynchronous events to be generated at the maximum precision of the underlying operating system and/or timer (specified in nanoseconds).

8. **Raw memory access** - Embedded applications frequently need direct access to specific regions of memory such as DMA, memory-mapped I/O, flash memory etc. The RTSJ has a specific interface to such memory areas without resorting to platform-specific native code.

5.1 Memory management

Many real-time systems only have a limited amount of memory. It is necessary to control how this memory is allocated so it can be used effectively. When there is more than one type of memory (with different access characteristics), it may be necessary to instruct the compiler to place certain data types at certain locations to increase performance and predictability. In RTSJ there are four types of memory:

- **Heap memory** - Collected by the garbage collector.
- **Stack memory** - Local variables collected when methods exit.
- **Immortal memory** - Never collected.
- **Scoped memory** - Available for collection when reference reaches zero.

To avoid dangling references, that is references to an object that has been collected, RTSJ requires assignment rules to be enforced, see table 7.

Table 7: RTSJ memory assignment rules

From Memory Area	To Heap Memory	To Immortal Memory	To Scoped Memory
Heap Memory	allowed	allowed	forbidden
Immortal Memory	allowed	allowed	forbidden
Scoped Memory	allowed	allowed	allowed to same scope or outer scope, forbidden if inner scope
Local Variable/Stack Memory	allowed	allowed	generally allowed

If an application violates an assignment rule a IllegalAssignmentError is thrown.

5.1.1 Memory areas

The RTSJ provides memory management which is not affected by garbage collection. It defines memory areas, some of which exist outside of the Java heap and never suffer garbage collection. RTSJ requires that the garbage collector can be preempted by real-time threads and that there should be a bounded latency for preemption to take place. The MemoryArea class is an abstract class from which all memory areas are derived. When a memory area is entered, all new object allocation is performed in that area. Subclasses of MemoryArea:

- **HeapMemory** - Allows objects to be allocated in the Java heap
- **ImmortalMemory** - Shared among all threads, never subjected to garbage collection and only freed when application terminates
- **ScopedMemory** - Memory area for objects that have a defined lifetime. Each scoped memory is associated with a reference count which keeps track of how many entities are currently using the area. The ScopedMemory class is an abstract class with several subclasses:

  - **VTMemory** - Allocations may take a **V**ariable amount of **T**ime
  - **LTMemory** - Allocation occur in **L**inear **T**ime

When real-time threads and asynchronous event handlers are created, they can be associated with memory parameters which specify:

- The maximum amount of memory a thread/handler can consume in a memory area
- The maximum amount of memory that can be consumed in immortal memory
- A limit of the rate of allocation from the heap (in bytes per second)

5.1.2 Nested memory areas

The real-time JVM needs to keep track of the currently active memory areas of a schedulable object. This can be achieved with a stack, so every time a schedulable object enters a memory area the identity of the area is pushed on the stack and when it leaves, it is popped of the stack. The stack can then be used to check for invalid memory assignments to and from scoped memory areas. For example, consider a schedulable object entering a scoped memory area A and then a scoped memory B. It then enters scoped memory area A again and creates an object referencing an object in B. When it then leaves A, the reference count is still not zero so memory is not reclaimed. It then leaves area B which leaves an object in area A with a dangling reference to area B. RTSJ avoids this problem by requiring each scoped memory area to have a single parent so it is not possible to enter an already entered memory area. It can however be necessary to switch between active memory areas. RTSJ implements this possibility with a "cactus" stack which allows the user to use the executeInArea method with a parameter of the heap or immortal memory resulting in a new memory stack being formed.

5.1.3 Heap memory

The JVM is responsible for managing the heap. Key problems are deciding how much space is required and when allocated space can be released. It can be handled in different ways:

1. Required the programmer to return the memory explicitly (C approach)
2. Require the JVM to monitor the memory and determine when it can no longer be accessed (out of scope)
3. Require the JVM to monitor the memory and release chunks which are no longer being used (garbage collection)

In a real-time application, those approaches will have an impact on the ability to analyse the properties of the application. Garbage collection may be performed either when the heap is full or by incremental activity. It can however have an impact on the response time of a time-critical thread.

See table 8 for a comparison between the stack and the heap in Java.

Table 8: Stack vs Heap

Stack	**Heap**
• Stores the variables of basic data types (int, boolean, references) local to a method.	• Stores all objects created from class definitions and requires garbage collection

5.1.4 Immortal memory

Immortal memory is outside of the heap and is not subject to garbage collection. Objects stored in immortal memory is never released until the application is terminated. There is only one ImmortalMemory area so the class is defined as final and only has one method: instance() to be able to access it. Immortal memory is shared among all threads in an application. The easiest way to allocate an object in immortal memory is to use the enter method of the MemoryArea class and pass an object implementing the Runnable interface, as follows:

```
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        /* Allocate objects.
         * All objects allocated here will be performed
         * in immortal memory*/
    }
}
```

5.1.5 Scoped memory

Scoped memory is also outside of the heap and not subject to garbage collection. Objects stored in scoped memroy have a well-defined life time. Schedulable objects can enter and leave a scoped memory area. Each scoped memory area has a reference count which indicates the number of times the scope has been entered. When the reference count goes from 1 to 0, the memory is immediately released and does not require garbage collection. In order to use scoped memory areas more efficiently it is necessary to estimate the amount of memory required. This can be done by using the **SizeEstimator** class. It can estimate the size of an object, but not the objects created during the constructor.

Scoped memory areas can be used in two modes: cooperatively or competitively.

- **Cooperatively** - Schedulable objects aim to be active in the same scoped memory area simultaneously. They use the area to communicate shared objects. The memory is reclaimed when they all leave the area. To share objects each schedulable object must have a reference to that object. A reference to such an object can only be stored in an object in the same scoped area or in an object in a nested scoped area. It cannot be stored in immortal or heaped memory area. Consequently, if there is no relationship between the schedulable objects, a schedulable object cannot pass a reference to an object it has just created to another schedulable object. RTSJ solves this with the use of "Portals". Each scoped memory area can have an object which acts as a gateway into that area.
- **Competitively** - Only one schedulable object is allowed in the memory area at one time. Goal is to make efficient use of memory as memory is reclaimed each time an object leaves the area.

5.1.6 Physical and raw memory

Today's computers have many different types of directly addressable memory available. Each type has its own characteristics that determine whether it is:

- **Volatile** - whether it maintains it's state when power is turned off
- **Writable** - whether it can be written at all, written once or written many times and whether writing is under program control
- **Erasable at the byte level** - if the memory can be overwritten and whether this is done at the byte level or whether whole sectors of memory need to be erased

- **Fast to access -** both for reading and writing
- **Expensive -** in terms of cost per byte and power consumption

Embedded real-time systems often support more than one type of memory. To allow the application to choose memory, RTSJ provides extensions of the MemoryArea class to provide physical memory classes. RTSJ also provides classes to allow the program to access raw memory locations to facilitate communication with I/O devices interfacing with the real world. These are:

- **LTPhysicalMemory**
- **VTPhysicalMemory**
- **ImmortalPhysicalMemory**

RTSJ supports access to physical memory via a memory manager and one or more memory filters. The goal of the memory manager is to provide a single interface with which the programmer can interact in order to access memory with a particular characteristic. The role of a memory filter is to control access to a particular type of physical memory. Memory filters may be dynamically added and removed, but there can only be one filter per memory type. The memory manager is unaware of the physical addresses of each type of memory, this is encapsulated by the filters. The filters also know the virtual memory characteristics that have been allocated to their memory type.

5.2 Time values and clocks

A real-time system often require:

- **Monotonic clock -** Progresses at a constant rate and is not subject to the insertion of extra ticks to reflect leap seconds.
- **Count down clock -** Can be paused, resumed and reset, e.g. the clock that counts down to the launch of a space shuttle.
- **Execution time clock -** Measures the amount of CPU time being consumed by a particular thread or object.

These clocks need a resolution potentially finer than the millisecond level. The RTSJ provides classes for more precise time values and clocks.

- **HighResolutionTime -** Time values with nanosecond granularity.
  - **AbsoluteTime -** Expressed as a time relative to some epoch. Epoch depends on the clock.
  - **RelativeTime**
- **Clock -** Abstract class from which all clocks are derived. The specification allows many different types of clocks, e.g. CPU execution-time clock or a real-time clock advancing monotonically.

The abstract methods **absolute(Clock)** and **relative(Clock)** allow time types that are relative to be re-expressed as absolute time and vice versa. The methods also allow the clocks associated with these values to be changed:

- **Absolute to absolute -** The value returned has the same millisecond and nanosecond component as the encapsulated time value.

- **Absolute to relative** - The value returned is the value of the encapsulated absolute time minus the current time as measured from the given clock parameter.
- **Relative to relative** - The value returned has the same millisecond and nanosecond component as the encapsulated time value.
- **Relative to absolute** - The value returned is the value of the current time as measured from the given clock parameter plus the encapsulated relative time.

Changing the clock is potentially unsafe since e.g. absolute time values are measured since a certain epoch. Different clocks may use different epochs.

If an event handler is associated with a timer it will be called when the timer fires. If not, nothing happens. When a timer has been created it can explicitly be destroyed, disabled (continues to count down but prevents firing) and enabled. If a timer is enabled after its firing time has passed, the firing is lost. The reschedule method allows the firing time to be changed and the start method starts the timer. Any relative time given in the constructor to a timer is converted to an absolute time. If an absolute time is given and the time has passed, the timer immediately fires.

5.3 Schedulable objects and schedulers

Standard Java offers no guarantees that the highest priority runnable thread will be the one executing. This is because a JVM may be relying on the host operating system to support it's threads where some might not support preemption. Java also only defines 10 priority levels and an implementation is free to map these priorities onto a more restricted host priority range if necessary.

RTSJ generalises the entities that can be scheduled from threads to the notion of schedulable objects. A schedulable object is one which implements the Schedulable interface. Each schedulable object must also indicate it's specific:

- **Release requirements** - When it should become runnable. Scheduling theories often identify three types of release:
  - **Periodic** - released on a regular basis.
  - **Aperiodic** - released at random.
  - **Sporadic** - released irregularly but with a minimum time between each release.

  All release parameters have a cost and a relative deadline time value. Cost is the amount of CPU time needed every release and the deadline is the time at which the current release must be finished.

  - **PeriodicParameters** - Also includes the start time for the first release and the time interval(period).
  - **SporadicParameters** - Includes the minimum inter-arrival time.
  - For aperiodic objects, it is possible to limit the amount of time the scheduler gives the min a particular period using **ProcessingGroupParameters**.
- **Memory requirements** - e.g. the rate at which the object will allocate memory on the heap.
- **Scheduling requirements** - e.g. the priority at which it should be started. Scheduling parameters are used by a scheduler to determine which object is most eligible for execution. The abstract class **SchedulingParameters** provides the root class for such parameters. RTSJ only defines one criteria which is based on priority, the higher the more important. **ImportanceParameters** allow an additional numerical scheduling metric to be assigned.

5.3.1 Schedulable object parameters

The attributes for a schedulable object can be summarised in the following:

- **ReleaseParameters** - The processing cost for each release, the cost, if the object is periodic or sporadic and it can also specify event handlers for when the deadline is missed or the processing resource consumed is greater than the cost specified. If no handlers are defined a count is kept on the number of missed deadlines.
  - **On each deadline miss** - If the thread has a deadline miss handler the value of descheduled is set to true and the deadline miss handler is released with a fireCount increased by missCount+1. If it does not have a miss handler, missCount is incremented.
  - **On each cost overrun** - If the thread has an overrun handler it is released or if the next release event has not already occurred (pendingRelease == 0) and the thread is Eligible-for-execution or Executing, the thread becomes blocked. If it is not eligible for execution or executing it is already blocked so the state transition is deferred. Otherwise a release has occurred and the cost is replenished.
  - **When each period is due** - If the thread is waiting for its next release and descheduled is true, nothing happens. If descheduled is false the thread is made eligible for execution, the cost budget is replenished and pendingReleases is incremented. Otherwise pendingReleases is incremented and if the thread is blocked-for-cost-replenishment it is made eligible-for-execution and rescheduled.
- **SchedulingParameters** - Empty class. Subclasses allow the priority to be specified and potentially it's importance. Importance is an additional scheduling metric to differentiate between objects with the same priority.
- **MemoryParameters** - The maximum amount of memory used by the object in the associated memory area. The maximum amount of memory used in immortal memory. A maximum allocation rate of heap memory.
- **ProcessingGroupParameters** - Allows several schedulable objects to be treated as a group and to have an associated period, cost and deadline.

If a schedulable object overruns it's cost budget it is immediately descheduled and will not be re-scheduled until either it's next release occurs or it's associated cost value is increased. If a schedulable object has sporadic parameters, that is it can be released at random times but with a minimum inter-arrival time, problems may occur that the scheduler must check for at run-time. Consider a sporadic event handler which is released by by a call to the fire method of it's associated event. Suppose the maximum length of it's queue is 3 and the implementation is simply keeping track of the time of the fire request. The minimum inter-arrival time is 2. Now say the queue is full and a new event occurs. There are then four possible solutions:

1. **Throw an exception** - ResourceLimitError is thrown on the offeding call of the fire method.
2. **Ignore the fire** - The call to fire the event is ignored.
3. **Replace the last release** - The last event is overwritten.
4. **Save the request** - The queue is lengthened.

The available actions on violation of the MIT are similar. Assume a new event occurs which violates the MIT.

1. **Throw an exception** - MITViolationException is thrown on the offending call of the fire method.
2. **Ignore the fire** - The call to fire the event is ignored.
3. **Replace the last request** - The last request is overwritten.
4. **Save the request** - The request is saved but the time is set to the MIT.

5.3.2 The Schedulable interface

All schedulable objects must implement the schedulable interface. The schedulable interface consists of the groups of methods:

- Methods which will communicate with the scheduler and will result in the scheduler adding or removing the schedulable object from the list of objects it manages or changing the parameters associated with the object.
- Methods which get or set the parameter classes associated with the schedulable object.
- Methods which get or set the scheduler.

5.3.3 Schedulers

The scheduler is responsible for scheduling it's associate schedulable objects. RTSJ supports priority-based scheduling via the **PriorityScheduler** (28 priority levels, fixed pre-emptive priority-based). This class is derived from the abstract **Scheduler** class which allows for implementation of e.g. Earliest-Deadline-First schedulers.

As more computers become embedded, there is however need for more flexible schedulers. In general, there are three ways of getting flexible scheduling:

1. **Pluggable scheduler** - The system provides a framework from within which different schedulers can be plugged in.
2. **Application-defined schedulers** - The system notifies the application every time an event occurs which requires a scheduling decision to be taken; the application then informs the system which thread should execute next.
3. **Implementation-defined schedulers** - An implementation is allowed to define alternative schedulers; typically this would require the underlying OS to be modified.

RTSJ adopts the implementation-defined scheduler approach. Applications can determine dynamically whether the real-time JVM on which it is executing has a particular scheduler. This is the least portable approach, as an application cannot rely on any particular implementation-defined scheduler begin supported. Priority-based scheduling is very flexible and by allowing the application to implement it's own scheduling policy on top of the PriorityScheduler a portable system can be constructed. The EDF scheduler illustrates this approach:

- All objects which are to be scheduled by the EdfScheduler run at one of three priority levels: low, medium or high.
- When schedulable objects are released, they are released at the high priority level and immediately call the reschedule method to announce themselves to the EdfScheduler.
- The EdfScheduler keeps track of the schedulable object with the closest absolute deadline and sets it's priority to the medium level.
- When a call is made to reschedule the EdfScheduler compares the calling object's deadline with that of it's closest deadline. If the caller has a closer deadline it is set to medium and the other object is set to low, otherwise it is set to low.
- After the end of each release, a schedulable object calls deschedule and the EdfScheduler sets it's priority to high and then scans it's list of schedulable objects to find the one with the closest deadline and sets it to medium.

5.3.4 Real-time threads

RTSJ defines two classes of real-time threads: RealtimeThread and NoHeapRealtimeThread. They are both schedulable objects and has all of the schedulable object parameters. By default, a real-time thread inherits the parameters of its parent. There are a few special methods for thread control specified in RTSJ:

- **waitForNextPeriod** - Suspends the thread until it's next release time unless the thread has missed it's deadline. Returns true whenever the thread is next released and if it is not a periodic thread, an exception is thrown.
- **deschedulePeriodic** - Causes the associated thread to block at the end of it's current release; it will then remain descheduled until schedulePeriodic is called.

One of the main weaknesses with standard Java, from a real-time perspective is that threads can be delayed by the garbage collector. RTSJ has solved this problem by allowing other memory areas than the heap. A NoHeapRealtimeThread can also be used and can be safely executed even when garbage collection is occurring. The constructors for a NoHeapRealtimeThread all contain references to specific memory areas where all allocation will occur. An unchecked exception is thrown if HeapMemory is passed. The start method is also redefined to check that the NoHeapRealtimeThread has not been allocated on the heap and that it has obtained no heap-allocated parameters. If these requirements are violated an unchecked exception is thrown.

To ensure that no-heap schedulable objects are not indirectly delayed by garbage collection requires:

- All no-heap schedulable objects should have higher priority than heap-using
- Priority ceiling emulation should be used for all shared synchronised objects
- All shared synchronised objects should have their memory requirements pre-allocated
- Objects passed in any communication should be primitive types passed by value or be from scoped or immortal memory.

5.4 Scheduling

Scheduling consists of three components:

- An algorithm for ordering access to resources (scheduling policy).
- An algorithm for allocating the resources (scheduling mechanism).
- A means of predicting the worst-case behaviour of the system when the policy and the mechanism are applied (schedulability analysis).

Example - The priority scheduler:

- **Policy -**
  - Supports base and active priority and orders the execution according to active priority.
  - Supports at least 28 unique priority levels.
  - Allows the programmer to define base priorities. Priorities may be changed at run time.
  - Supports priority inheritance or priority ceiling emulation inheritance for synchronised objects.
  - Assigns the schedulable object the active priority of the higher of it's base priority and any priority it inherits.

- **Mechanism -**
  - Supports pre-emptive priority-based dispatching of schedulable objects - processing resource is always given to the highest priority runnable object.
  - Does not define where in the run queue a pre-empted object is placed.
  - Places a blocked schedulable object which becomes runnable at the back of run queue associated with it's priority.
  - Places a thread which performs a yield operation at the back of the run queue of it's associated priority.
- **Schedulability analysis -**
  - Requires no particular analysis to be supported.
  - Default analysis assumes an adequately fast machine.
  - PriorityScheduler also adds three methods to query maximum, normal and minimum priority levels.

$$P_{normal} = \left\lceil P_{min} + \frac{P_{max} - P_{min}}{3} \right\rceil$$

5.4.1 Fixed priority scheduling (FPS)

FPS requires:

- Statically allocating schedulable objects to processors.
- Ordering the execution of schedulable objects on a single processor according to a priority.
- Assigning priorities to schedulable objects at their creation time. It is usual to assign priorities to the relative deadline of the schedulable object: the shorter the deadline, the higher the priority.
- Priority inheritance when accessing resources.
- Pre-emptive priority-based dispatching of processes - the processing resource is always given to the highest priority runnable schedulable object (allocated to that processor).

There are many ways for analysing whether a fixed priority-based system will meet it's deadlines, but the most flexible is response time analysis. After a schedulable object is started, it waits to be released. When released it performs some computation and then waits to be released again. The release profile defines the frequency with which it's releases occur, they may be time or event triggered. A schedulable object also has an associated cost (compute time) as well as a deadline specifying when the object has to complete the computation associated with each release. There is a metric which indicates a schedulable objects contribution to the overall functionality of the system called **Value**. It may be:

- A coarse indicator - e.g. safety critical, mission critical, non critical.
- A numeric value giving a measure for a successful meeting of a deadline.
- A time-valued function which takes the time at which the object completes and returns a measure of the value.

In real-time systems the priority of a process is derived from its temporal requirements, not its importance to the system.

5.4.2 Schedulability analysis

A key characteristic of schedulability analysis is whether the analysis should be performed off-line or on-line. For safety critical systems, off-line analysis is essential. Other systems that do not have such stringent timing requirements, on-line analysis may be appropriate. RTSJ provides a framework for on-line analysis of priority-based systems for single processor systems. The specification also allows a real-time JVM to monitor resources being used and to fire AEH if the resources go beyond specification provided by the programmer.

There are different ways to analyse schedulability, among them utilisation based analysis. It is a simple sufficient test but not necessary test.

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$$

Here N = number of processes, C = worst-case computation time and T = minimum time between process releases. There is a set utilisation bound for this analysis as seen in table 9.

Table 9: Utilisation bounds

N	Utilisation bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%
∞	69.3%

There is also the Response-time analysis which is necessary and sufficient. Here task i's worst-case response time R_i is calculated as $R_i = C_i + I_i$, where C is worst-case computation time and I is the interference time from higher priority tasks. During R, each higher priority task j will execute a number of times: Num releases = $\left\lceil \frac{R_i}{T_j} \right\rceil$, T = minimum time between process releases. So total interference is given by $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$. This results in

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where hp(i) is the set of tasks with higher priority than task i. The problem is solved by forming a recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

The set of values $w_i^0, w_i^1, w_i^2, w_i^n$ is monotonically non-decreasing and the solution has been found when $w_i^n = w_i^{n+1}$. w_i^0 must not be greater than R_i.

The worst-case execution time can be found either by measurement or analysis. Measurements are difficult because you can not be certain you've observed the worst case. Analysis is difficult because you need an effective model of the processor. Most analysis techniques involve two distinct activities:

- The first takes the process and decomposes its code into a directed graph of basic blocks. These blocks represent straight-line code.

- The second component takes the machine code corresponding to a basic block and uses the processor model to estimate the worst-case execution time. Once the times for all basic blocks are known, the graph can be collapsed.

General guidelines are:

1. All processes should be schedulable using average execution time and average arrival rates
2. All hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft)

If a process is suspended because of priority inversion, the priority model is, in some sense, being undermined. If a process is waiting for a lower-priority process, it is said to be blocked (B_i). Then R_i and w_i^{n+1} become

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$
$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Sometimes, arrival patterns and computation times cannot be known a priori. Then an on-line analysis is required. The main task of such a scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment. EDF is a dynamic scheduling scheme which can suffer from "transient overloads" where it is possible to get cascade effect in which each process misses it's deadline but uses sufficient resources to result in the next process missing it's deadline. To counter such effects, many on-line schemes have two mechanisms:

1. An admissions control module that limits the number of processes that are allowed to compete for the processors
2. An EDF dispatching routine for those processes that are admitted

An ideal admissions algorithm prevents the processors from becoming overloaded so the EDF routine works effectively. For the admissions algorithm to be able to select processes to admit the relative importance of the processes must be known. It is usually achieved by assigning **value**. Values can be classified as:

- **Static** - the process always has the same value whenever it is released.
- **Dynamic** - the process' value can only be computed at the time the process is released.
- **Adaptive** - here the dynamic nature of the system is such that the value of the process will change during execution.

5.4.3 Simple process model

- The application is assumed to consist of a fixed set of processes
- All processes are periodic, with known periods
- The processes are completely independent of each other
- All system overheads are ignored
- All processes have a deadline equal to their period

- All processes have a fixed worst-case execution time

One common way of implementing hard real-time systems is to use a cyclic executive. The design is concurrent but the code is produced as a collection of procedures. Procedures are mapped onto a set of minor cycles that constitute the major cycle. The minor cycle dictates the minimum cycle time and the major cycle the maximum cycle time. No actual processes exist at run-time, each minor cycle is just a sequence of procedure calls. The procedures share a common address space and can pass data between themselves. This data does not need protection because concurrent access is not possible. All "process" periods must be a multiple of the minor cycle time. Problems with cyclic executives:

- Difficult to incorporate processes with long periods; the major cycle time is the maximum period that can be accommodated without secondary schedules
- Sporadic activities are difficult (impossible) to incorporate
- The cyclic executive is difficult to construct and difficult to maintain
- Any process with a sizeable computation time will need to be split into a fixed number of fixed size procedures.
- More flexible scheduling methods are difficult to support
- Determinism is not required, but predictability is.

5.4.4 Asynchronous events and event handlers

To respond to asynchronous events RTSJ defines Asynchronous Event Handlers (AEH) as schedulable objects to be able to respond within a given deadline. A real-time JVM will usually dynamically associate an event handler with a real-time thread when the handler is released for execution. To avoid this overhead it is possible to associate an event handler with a real-time thread permanently. This is done by the class BoundAsyncEventHandler. A BAEH is permanently associated with a dedicated server real-time thread. A server thread can be considered to be a daemon thread. This means that when all user real-time threads are terminated the program will terminate, but where events are bound to happenings in the environment and timers, the program may not execute as intended.
Each AsyncEvent can have one or more handlers and the same handler can be associated with more than one event. When the event occurs, all handlers associated with it are released for execution according to their SchedulingParameters. Each AEH has count of the number of outstanding firings (fireCount). When an event is fired, the count is atomically incremented. The handler is then released. A set of protected methods allow the fire count to be manipulated. They can only be called by creating a subclass and overriding the handleAsyncEvent method. If a Runnable object has been supplied with the constructor, the run method of that object is executed when handleAsyncEvent is executed. The run method of the AEH class itself is the method that will be called by the underlying system when the object is first released. It will in turn call handleAsyncEvent repeatedly whenever the fire count is greater than zero.
Asynchronous events can also be associated with interruptions or POSIX signals (if supported by the OS) or linked to a timer. The timer will cause the event to fire when a specified time (relative to a clock) expires. Can be one shot or periodic.

Asynchronous transfer of control in RTSJ:

- A schedulable object must explicitly indicate that is is prepared to allow an asynchronous transfer of control (ATC) to be delivered. By default, ATC is deferred.
- The execution of synchronised methods and statements always defer delivery of an ATC.

- An ATC is a non-returnable transfer of control
- An AsynchronouslyInterruptedException AIE class define the ATC event
- A method prepared to allow an AIE must declare so in it's throw list
- The Interruptible interface provides the link between the AIE class and the object executing an interruptible method

Systems may be either time triggered or event triggered:

- **Time triggered** - In a time triggered system, the controller is activated periodically; it senses the environment in order to determine the status of the real-time objects it is controlling. E.g. a robot controller may determine the position of a robot via a sensor and decide that it must cut the power to a motor.
- **Event triggered** - In an event triggered system, sensors in the environment are activated when real world object enter various states. The events are signaled to controllers via interrupts. E.g. a robot may trip a switch when it reaches a certain position. Event triggered systems are more flexible whereas time triggered systems are more predictable. In both the time triggered and event triggered systems, controllers are usually represented as a thread.

There are occasions when threads are not appropriate:

- When the external objects are many and their control algorithms are simple and non-blocking
- The external objects are inter-related and their collective control requires significant communication and synchronisation between the controllers

An alternative to thread-based programming is event-based programming. Each event has an associated handler; when events occur, these are queued and one or more server threads takes an event from the queue and executes the associated handler. There is no need for explicit communication between the handlers as they can read and write from shared objects without contention. The disadvantage of controlling all external objects with event handlers include:

- It is difficult to have tight deadlines as a long-lived and non-blocking handler must terminate before the server can execute any newly arrived high-priority handler.
- It is difficult to execute the handlers on a multiprocessor system as the handlers assume no contention for shared resources.

5.4.5 Asynchronous transfer of control

An asynchronous transfer of control (ATC) is where the point of execution of one schedulable object is changed by the action of another. Consequently a schedulable object might be executing one method and then through no action of its own execute another method. Reasons to use ATC:

1. **Error recovery** - to support coordinated error recovery between real-time threads. Where several threads are working on a problem, an error detected by one thread may need to be quickly and safely communicated to other threads.
2. **Mode changes** - where changes between modes are expected but cannot be planned. The processes must be quickly and safely informed that the mode in which they are operating has changed and they now must take other actions.

3. **Scheduling and interrupts** - Scheduling using partial/imprecise computations - there are many algorithms where the accuracy of the results depend on how much time can be allocated to their calculation. Also, in a general interactive computing environment, users often wish to stop the current processing because they have detected an error condition.

Such situations might traditionally be addressed with a polling approach and aborting the application if required. A problem with this is that polling is slow. One approach to ATC is to destroy the thread and allow another to recover. Destroying a thread is however expensive and is often an extreme reaction.

When a real-time thread is interrupted an asynchronous exception is thrown at the thread rather than the thread having to poll for the interruption. The transfer of control may be triggered by an asynchronous event such as a timer, a call to interrupt() or a call to AsynchronouslyInterruptedException.fire(). With this facility, the execution of code within one schedulable object can be altered from one method to another via actions of another schedulable object. To avoid thread-related issues such as deadlock, there are specific rules for ATC.

The main problem with AE is how to program safely for their existence. Most exception handling mechanisms have exception propagation within a termination model. This means that after an exception has been handled, the application continues to execute within the context where the handler was found. The RTSJ solution to this is to require that all methods which are prepared to allow ATC, place the AIE exception in their throw list. RTSJ calls such methods AI-methods (Asynchronously Interruptible). If a method does not do this, the exception is not delivered but held pending until the thread is within a method with the throw clause. Hence, code written without regard to ATC can be executed safely. For ATC to handled safely, RTSJ requires that:

- ATCs are deferred during the execution of synchronised methods or statements
- RTSJ calls these sections of code and methods which are not AI methods ATC-deferred sections.
- An ATC can only be handled from within code that is an ATC-deferred section. This is to avoid an ATC handler from being interrupted by another ATC being thrown.

The use of ATC requires declaring an AIE, identifying methods which can be interrupted and signaling an AIE to a schedulable object. Calling interrupt() in the RealtimeThread class throws the systems generic AIE.

5.5 Synchronisation

Ordinary Java provides mutually exclusive access to shared data via monitor constructs. These suffer from priority inversion and the solution is priority inheritance. The RTSJ explicitly supports two methods of priority inheritance: simple priority inheritance and priority ceiling emulation inheritance.
If real-time threads want to communicate with non real-time threads, the garbage collector must be considered. If a non real-time and a real-time thread enter into a mutual exclusion zone the actions of the non real-time thread results in garbage collection. The real-time thread then pre-empts the garbage collector but is unable to enter the zone and must wait for the garbage collector and the non real-time thread to exit the zone. To avoid such behaviour, RTSJ provides wait-free non blocking classes to facilitate this communication:

- **WaitFreeWriteQueue** - A bounded buffer, the read operation is synchronised, write is not.
- **WaitFreeReadQueue** - A bounded buffer, the write operation is synchronised, read is not. A reader can request to be notified when data arrives (AE).

5.5.1 Priority inversion

Priority inversion is when a lower priority thread blocks a higher priority thread because of a synchronised statement or method. Priority inversion can occur when a schedulable object is blocked waiting for a resource. The solution to priority inversion is called priority inheritance. Basically, if a high priority thread tries to access an object which is locked by a lower priority thread, the low priority thread inherits the high priority so it can release the object more quickly. RTSJ requires the following:

- All queues maintained by the system need to be priority ordered (e.g. the queue of schedulable objects waiting for an object lock)
  - Where there is more than one schedulable object with the same priority the order should be first in first out.
  - Similarly, the queues resulting from calling the wait method in the Object class should be priority ordered.
- Facilities for the programmer to specify different priority inversion control algorithms. By default, the RTSJ requires simple priority inheritance to occur whenever a schedulable object is blocked waiting for a resource.

A certain type of priority inheritance is priority ceiling emulation:

- Each thread has a base priority
- Each resource has a static ceiling value defined, this is the maximum priority of the threads that use it
- A thread has an active priority that is the maximum of the base priority and the ceiling value of any resource it has locked
- As a consequence, a thread will only suffer a block at the very beginning of its execution
- Once the thread starts executing, all the resources must be free, if they're not then some other thread would have an equal or higher priority and the thread would be suspended.
- If a thread calls a synchronised statement or method in an object the real-time VM will check the active priority of the caller. If it's priority is higher than the ceiling priority an unchecked CeilingViolationException is thrown.

6 Java on the web

Java was originally intended to be delivered over the internet to your computer but the internet was too slow back in the days. The idea was that the user pays per use of the program which is not really a viable model anymore.

6.1 Web applications

A web application is a series of web pages generated in response to user requests. Web applications use a client/server architecture. The web browser forms the client and the web server forms the service provider. The web server runs on the server using web server software; a widely used web server is the Apache server. Most web servers also use a Database Management System (DBMS). Two popular ones are Oracle and MySQL. There are two types of web pages, static and dynamic:

- **Static web page -** Doesn't change in response to user input. If the user clicks on something, a request is sent to the server and the server returns a new HTML file.
- **Dynamic web page -** Changes in response to user input. A dynamic web page is an HTML document generated by the web server in real time. When the user clicks something, a request is sent to the server and the server passes it on to the application. The application then generates an HTML file in response and that is sent back to the client.

There are three modern approaches to Java web applications:

1. **Servlet/JSP -** Servlets store the Java code that does the server side processing and JavaServer Pages (JSP) store the HTML that define the user interface.
    - The running HTML typically contains links to CSS and JavaScript.
    - The Servlet/JSP API is low level - giving high control but requiring much coding.
    - Foundation for the other methods
2. **JavaServer Faces (JSF) -** Newer technology supposed to replace Servlets and JSP. Provides an API that provides more fully developed modules for developers.
3. **Spring Framework -** Higher level API doing more pre-work for the developer but can still provide a high level of control over the HTML/CSS/JavaScript returned to the client.

6.2 Servlets and JSP

The architecture of the Servlet/JSP approach contains three layers:

1. **Presentation layer/User interface layer -** Consists of HTML pages and JSPs. Creates the "look and feel" of the user interface.
2. **Business rule layer -** Uses Java Servlets to control the flow of the application. The servlets may be programmed to call other Java classes to store or retrieve data from a database or send data. Servlets may also forward results to a JSP or another servlet. Servlets may also take advantage of JavaBeans, a set of classes which can store and process data in different ways. For example a JavaBean can be used to define an invoice object .
3. **Data access layer -** Works with the data layer of the application on the servers disk. Data is typically stored in a database (MySQL or SQL) but can also be stored in other files.

Directories required for web application using Servlets and JSP:

- **WEB-INF -** Contains a web.xml file for the application which tells the web server how this application should be deployed on the server. Can be used to store data files or other files the user shoud not have access to.
- **WEB-INF\classes -** Root directory for all Java class fiels not stored in JAR format (Java class zip files). This includes servlets.
- **WEB-INF\lib -** Contains the JAR files for Java class libraries needed by the web application, can be sourced from a third party. In Apache (tomcat6) there is also a lib directory in the root tomcat directory which stores class files available to ALL web applications.
- **META-INF -** Typically contains a context.xml file which can configure the applications context.

- **Root directory -** Contains all other directories for the web application. For Apache - web application roots are located under webapps.

The most popular web servers for Java are:

- **Tomcat**
  - Server/JSP engine called Catalina with a web server called Coyote
  - Free and open-source
  - Most popular web server for Java applications
- **Glassfish**
  - Complete Java EE application server
  - Free and open-source
  - More features than tomcat but requires more resources

HTTP uses GET and POST methods which are used in the following scenarios:

- **GET -**
  - Request reads data from the server
  - Request can be executed multiple times without causing any problems
- **POST -**
  - Request writes data to the server
  - Executing the request multiple times may cause problems
  - Do not want parameters in the URL for security reasons such as users not bookmarking a URL with parameters
  - Need to transfer more than 4kB of data

6.2.1 Creating and mapping Servlets

Two steps are required to create a servlet:

1. Code the class for the servlet
2. Map the created servlet class to a URL - prior to servlet 3.0 specification this had to be done in the web.xml file but now one can use the @WebServlet notation.

To code a servlet to deal with HTTP GET and POST requests a class must extend the GenericServlet class or the HttpServlet class. Generally speaking, the HttpServlet class is extended and requires the import of javax.servlet.http packages. Then the doGet and goPost methods of HttpServlet need to overridden.

Mapping can be done by the @WebServlet notation, e.g. ”@WebServlet(”/name”)” before the declaration of the servlet and after importing javax.servlet.annotation.WebServlet. It can also be done in the web.xml file by the following code snippets:

```
<!-- definitions for the servlet. The servlet is located in WEB-INF/classes/murach/email/ under the
    root folder. -->
<servlet>
   <servlet-name>EmailListServlet</servlet-name>
   <servlet-class>murach.email.EmailListServlet</servlet-class>
</servlet>

<!-- mapping the servlet. It is mapped twice to allow different URLs. -->
<servlet-mapping>
   <servlet-name>EmailListServlet</servlet-name>
   <url-pattern>/emailList</url-pattern>
</servlet-mapping>
<servlet-mapping>
   <servlet-name>EmailListServlet</servlet-name>
   <url-pattern>/email/*</url-pattern>
</servlet-mapping>
```

To test a servlet, use a HTTP GET and POST request:

```
<a href ="emailList?action=join">Display Email Entry Test</a>

<!-- Can also enter the URL into the browser address bar. For servers running locally:
http://localhost:8080/ch05email/emailList?action=add&firstName=John -->
```

6.2.2 Getting parameter values into servlets

There are two methods of the HttpServletRequest class that can be used to get parameters: getParameter(String param) and getParameterValues(String param). To get text a user has entered in a text box, use the getParameter and specify the name of the text box. To get values from a drop down list with the possibilites of many different values, use the getParameterValues and specify the name of the list.

To access a file (to write values or retrieve values or such) you sometimes need the real path to the file. This can be done by calling the GenericServlet method getServletContext and then calling getRealPath on the ServletContext. A ServletContext object can also be used to work with global variables, read global initialisation parameters and to write data to log files.

6.2.3 Get and set request object attributes

To store an object in the HttpServletRequest object, use the setAttribute(String name, Object o). To get them, use the getAttribute(String name); when getting an attribute, the type must be cast to the wanted type as it returns an ”Object”.

6.2.4 Forwarding a request object

The ServletContext class contains the method getRequestDispatcher(String path). This returns a RequestDispatcher which can be used to forward ServletRequests by calling forward(ServletRequest request, ServletResponse response). Both the request and response are forwarded to easily access the response. See example below:

```
import javax.servlet.http.*;
```

```
HttpServletRequest request = new HttpServletRequest();
HttpServletResponse response = new HttpServletResponse();
String url = "/index.html";

getServletContext().getRequestDispatcher(url).forward(request, response);
```

A response can then be redirected by calling the sendRedirect(String path) method of the HttpServletResponse class. Often it requires an absolute URL but can be used with relative paths as well. To redirect to a completely different web server, supply a full URL to that server.

6.2.5 Validating data on the server

Validating data on the server can be useful for e.g. login pages or account registration. The data is then validated to see that the user has entered a correct email address for instance. This can be done by letting the webpage/HTML page display an optional message after input is entered. After the user clicks on a button, the doPost method of the servlet is run. By using the getParameter method described in section 6.2.2 to retrieve the parameters, the data in them is checked for validity with ordinary Java code and if it is invalid, a message is set and sent back to the HTML page which displays the message.

6.2.6 Initialisation of parameters and servlets

Parameters intended to be available to all servlets as well as servlets themselves need to be initialised when the web application is started. This is done in the web.xml file. Common XML elements for working with initialisation are:

```
<context-param>
<servlet>
<servlet-name>
<servlet-class>
<init-param>
<param-name>
<param-value>
```

To retrieve an initialisation parameter available to all servlets one uses the getServletContext method and then getInitParameter(String name) on that ServletContext object. To retrieve a initialisation parameter available to a single servlet one uses the getServletConfig() to retrieve a ServletConfig object and then use the same method as before.

6.2.7 Servlet methods

A servlet engine (such as tomcat) only creates one instance of a servlet when the engine starts or when the servlet is first requested. Therefore the init method is only called once. After the servlet has been created, each subsequent request for the servlet spawns a thread that calls the servlets service method. The service method checks what method is specified in the HTTP request and calls either the doPost or the doGet method. Because each request for the servlet spawns a new thread, instance variables within a servlet should not be used, as it is not threadsafe. There is however a way of getting around this: by using sessions. Sessions are used to track user by using something called a cookie. A per-session cookie is stored in the client's browser and passed along with every HTTP request. Another way of achieving the same tracking is by using URL encoding where the unique session id is included in the URL. This is very unsafe though as it can lead to hijacking of sessions, so it's better to use cookies. To find out what user the servlet is treating one can use the getSession method from the HttpServletRequet object. If the session object does not exist for this client, a new one is created. Session objects also have attributes which are set, gotten and removed by the usual methods. This can be used to associate objects with a specific session, e.g. an item cart for e-shopping. When an attribute is set in a session object it is available until one of the following occurs:

- The user closes the browser page
- The session times out
- The use of removeAttribute

A session object has some methods for identifying it and retrieving it's attributes, one of them begin getId() which returns a unique id for the session. There is also the isNew method which returns a true value if the client is accessing the site for the first time or if cookies are disabled. As stated before, instance variables are not threadsafe as a thread is created per request. To make accessing instance variables threadsafe do the following:

```java
Cart cart;
final Object lock = request.getSession().getId().intern();
synchronized(lock) {
    cart = (Cart) session.getAttribute("cart");
}
```

Sometimes things nevertheless go wrong and it can be useful to print some debug data. There are two methods of the HttpServlet class which allow the user to log debug data: log(String message) and log(String message, Throwable t). The latter of the two also prints the stack trace for an exception.

6.2.8 JavaBeans

A Java Bean is a Java class that:

- Provides a zero argument constructor
- Provides get and set methods for all of its private instance variables
- Implements the Serializable or Externalizable interface. The Serializable interface is a tagging interface that indicates that the class must implement set and get methods.

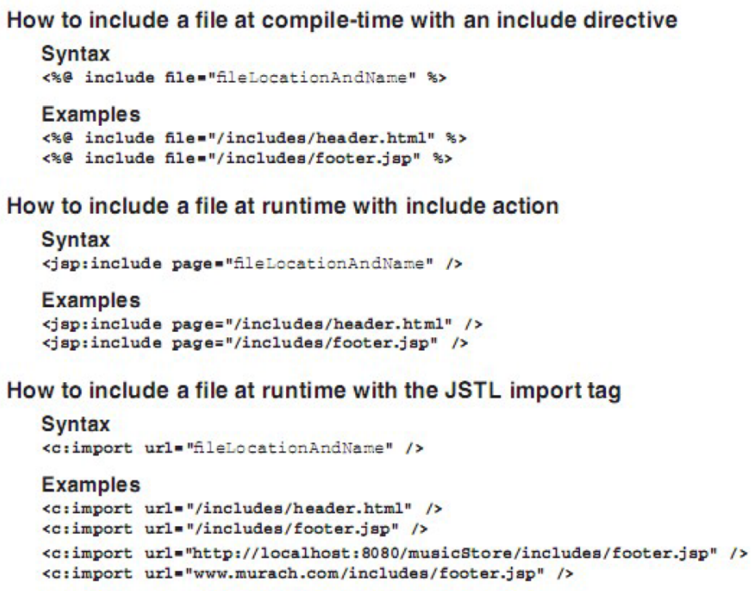
A Java Bean can look like the following:

```java
import java.io.Serializable;
// Implements Serializable
public class User implements Serializable {

    private String firstName;
    private String lastName;

    // A zero argument constructor
    public User() {
        firstName = "";
        lastName = "";
    }

    //Provides set functions for it's private variables
    public void setFirstName(String name) {
        firstName = name;
    }

    public void setLastName(String name) {
        lastName = name;
    }

    //Provides get functions for it’s private variables
    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

}
```

The main benefit of using JavaBeans is that they can be used with Expression language and standard JSP tags (see section 6.2.10. This is done as follows: <jsp:useBean id=”nameofBean” scope=”request” class=”nameofclass”/ >. Then the JavaBean is accessed by the following: <span><jsp:getProperty name=”user” propert=”email”/ ></span>. The advantages of using standard JSP tags is that they create a JavaBean if it does not already exist. The tags also provide a way of setting the properties.

6.2.9 Including files in JSP

Sometimes the same block of code is used in several JSPs. To simplify maintenance, reduce number of lines of code to write and reduce error, the duplicate code can be stored in a separate file and be included in the JSP file. There are three distinct techniques to include files as explained in figure 5.



How to include a file at compile-time with an include directive

Syntax

```
<%@ include file="fileLocationAndName" %>
```

Examples

```
<%@ include file="/includes/header.html" %>
<%@ include file="/includes/footer.jsp" %>
```

How to include a file at runtime with include action

Syntax

```
<jsp:include page="fileLocationAndName" />
```

Examples

```
<jsp:include page="/includes/header.html" />
<jsp:include page="/includes/footer.jsp" />
```

How to include a file at runtime with the JSTL import tag

Syntax

```
<c:import url="fileLocationAndName" />
```

Examples

```
<c:import url="/includes/header.html" />
<c:import url="/includes/footer.jsp" />
<c:import url="http://localhost:8080/musicStore/includes/footer.jsp" />
<c:import url="www.murach.com/includes/footer.jsp" />
```

Figure 5: Including files in JSP

The first method has the advantage of being quick at run-time since the file has been included at compile-time. The disadvantage is that to update code the JSP file must be recompiled possibly forcing the servlet engine to restart or the web application to be recompiled. The second method lets the servlet engine always use the latest version of the included file. The disadvantage is that it is not very efficient at run-time and adds extra time. The last approach allows the engine to always use the latest version as well as allowing to include code from other applications or even other web servers. The disadvantage is still that is is not very efficient. Which method to choose depends on the application but many newer servlet engines automatically detect changes to included files and in these cases it is obviously advantageous to include them at compile time. However if the information in the files is known to change frequently a run-time method is of course preferred.

6.2.10 JSP tags

JSP tags allows embedding of Java statements directly into JSP. The result is basically a mixture between HTML and Java. There are five JSP tag types as seen in table 10.

Table 10: JSP tag types

—**Tag**	—**Name**	**Purpose**
—< %@ % >	—JSP directive	To set conditions that apply to the entire JSP
—< % % >	—JSP scriptlet	To insert a block of Java statements
—< %– % >	—JSP expression	To display the string value of an expression
—< %– –% >	—JSP comment	To tell the JSP engine to ignore code
—< %! % >	—JSP declaration	To declare instance variables and methods for a JSP

6.3 Java Applets

A Java applet is a part of a web page and occurs inside a web browser running it's own JVM. When the web page closes, so does the JVM. Generally, the applet does not have access anywhere outside it's own context to provide better security. However, Applets have proven to be insecure anyway, so they are now legacy (but implemented as Servlets nowadays).

- An applet is embedded in a web page and operates in it's context
- An applet must be a subclass of the java.applet.Applet class which provides the standard interface between the Applet and the browser interface
- The Swing package provides a subclass of the Applet class called javax.swing.JApplet which should be used for any applets that use the Swing package to construct their GUIs

To program an applet follow the following steps:

1. Create a Java source file
2. Compile the source file
3. Create a HTML file using a text editor. This file should include the Applet (the compiled source file). Can be done as follows: ¡APPLET CODE="nameOfApplet.class" WIDTH=150 HEIGHT=25¿ ¡/APPLET¿
4. Run the applet (HTML file) using a compatible browser

Applets are controlled by the following methods:

- **init()** - Called once when the applet is created. Performs initialisation.
- **start()** - Called when the applet becomes visible.
- **stop()** - Called when the applet becomes invisible.
- **destroy()** - Called to give an applet a last chance to clean up before it is removed.

Applets are either privileged or sandbox:

- **Privileged applets** - Can run outside their security sandbox and have extensive ability to access the client.
- **Sandbox applets** - Runs in a security sandbox that allows only a set of safe operations. Cannot load native libraries. Cannot change the security manager and cannot read certain system properties such as the java.class.path or user.home.

Applets that are not signed are automatically considered sandbox applets. Applets that are signed by a certificate from a recognised authority can either run in sandbox mode or request access to run as a privileged applet.