# Table of Contents

[Task 1: Reflection and output file generation 2](#_Toc481016811)

[Task 2 – Designing and implementing a threaded application 3](#_Toc481016812)

[Car class 3](#_Toc481016813)

[Checkpoint class 3](#_Toc481016814)

[Race class 4](#_Toc481016815)

[Main class 4](#_Toc481016816)

[Task 3 – Java Clone Mechanism 5](#_Toc481016817)

[What is Clone Mechanism 5](#_Toc481016818)

[Why it’s required by Java language 5](#_Toc481016819)

[How it’s implemented in Java (discuss & illustrate) 5](#_Toc481016820)

[Options available for implementing cloning when cloning inheritance hierarchies 5](#_Toc481016821)

[Alternatives to Java clone mechanism 6](#_Toc481016822)

[References 6](#_Toc481016823)

[Appendix 7](#_Toc481016824)

[Task1 files 7](#_Toc481016825)

[Team.java 7](#_Toc481016826)

[Reflection.java 9](#_Toc481016827)

[Task 2 Files 11](#_Toc481016828)

[Race.java 11](#_Toc481016829)

[Main.java 12](#_Toc481016830)

[Checkpoint.java 13](#_Toc481016831)

[Car.java 13](#_Toc481016832)

[Figure 1: No file before Refelection 2](#_Toc481016833)

[Figure 2: reflectionFile.java content 2](#_Toc481016834)

[Figure 3: file created 2](#_Toc481016835)

[Figure 4: code executed. threads starting 4](#_Toc481016836)

[Figure 5: drivers finishing in different order 4](#_Toc481016837)

# Task 1: Reflection and output file generation

For the generation of a .java from a source file using the reflection API, we will use a class that creates team objects. The code for the source file is in the appendix.

The Reflection class will use the java.lang.reflect class, and an object of class Team will be created to access the different variables and methods available in the original source file.

The java.io.File and java.io.PrintStream classes will also be used to create a file and print to it. For this instance, the file will be named “reflectionFile.java”, and since no path has been declared, the file will be created in the Project folder.

The source file contains the following information:

* package name
* class name
* variable names and their types
* constructor
* methods (void and return types)

When running the Reflection class, the program will generate the output file and will contain the above information, excluding arguments and logic within the methods.

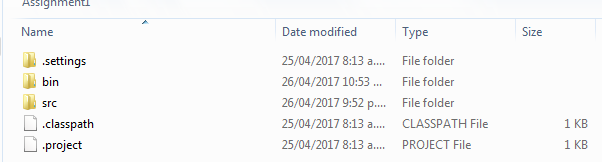


Figure : No file before Refelection

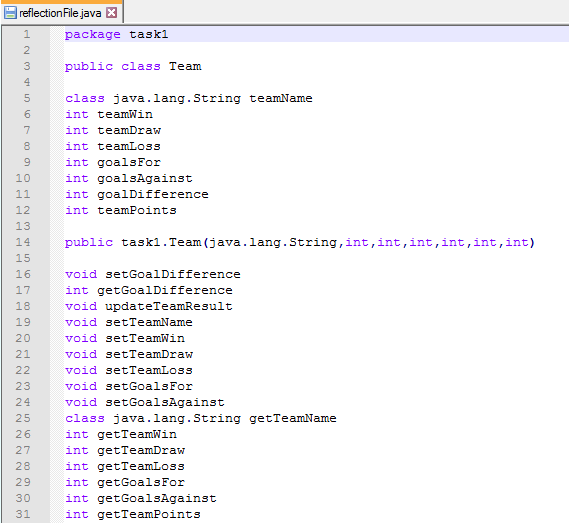


Figure : reflectionFile.java content

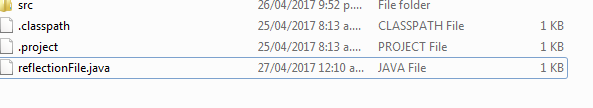


Figure : file created

# Task 2 – Designing and implementing a threaded application

The scenario for the threaded application is a Rally race, where several drivers race each other. They start at the same time, and go through checkpoints, where they re-fuel their cars, before going to the next checkpoint until they cross the finish line.

The program will get each thread (driver) to start one after another, and complete the lap in a random time, generated by the Car.raceLap() method.

When they get to the checkpoint, the re-fuel. However only one driver can re-fuel at a time, and for a random number of seconds (Checkpoint.reFuel()). Only when that driver has left than the next one can re-fuel. Hence for the Checkpoint.reFuel() method to be a synchronized method. This ensures that the thread has a lock, and until the car has re-fuelled and left, the lock is not relinquished. When it does, the notifyAll() method will ensure that the other threads are awakened for them to perform their task, in that case re-fuel.

Every time someone crosses the finish line, they are ranked, and the rank value is incremented for the next driver.

The application will have 4 classes, which are:

* Car
* Checkpoint
* Race
* Main

## Car class

It deals with the creation of the Car object, although for the running of the application it looks better with the name of pilots instead. Its methods are:

raceLap() 🡪 how long to race a lap. Throws InterruptedException as it will not start until it has re-fuelled, which happens only after preceding car has done so and left.

Run() 🡪 method for threads

## Checkpoint class

Simple class that creates an object rdm to store a random number between 2 and 3 seconds (0+1000 and 2000+1000 milliseconds). The refuel() method is used in the Car.run() method to say how long the car re-fuelled for, or how long the thread waits before starting again.

## Race class

Contains the following methods:

getReadyToRace() 🡪 throws InterruptedException and gets threads to start “at the same time”.

startRace() 🡪 calls notifyAll() to awaken threads.

crossFinishLine() 🡪 increments rank value when cars cross finish line

## Main class

Contains the Main method, to create all instances and threads for running the program. Such objects and methods are:

The Race and Checkpoint object, along with the multiple threads.

The methods called are:

start() 🡪 to start each thread

race.startRace() 🡪 Calls startRace from the Race class.

As can be seen below, this is what happens when the program is run:

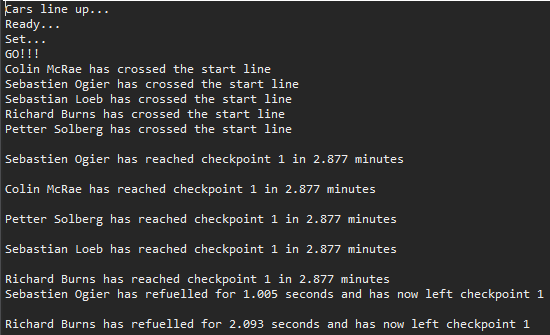


Figure : code executed. threads starting

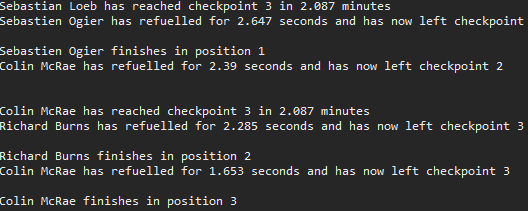


Figure : drivers finishing in different order

# Task 3 – Java Clone Mechanism

## What is Clone Mechanism

The Clone Mechanism in Java is a method for duplicating objects. The Clone() method provides the functionality of duplicating an object, while the assignment operator can only duplicate references (clone (Java Method), 2017). It does so by implementing the Cloneable interface, and since it cannot know in advance what is in the class it is trying to replicate, it does a field-by-field copy.

The clone object created by the cloning mechanism and the original object are two separate objects in the memory. It clone is an exact copy of the original object.

## Why it’s required by Java language

The clone() method copies the fields and values of an object to another. Although this requires nearly twice the space since there are 2 objects, this saves the extra processing task for creating the exact copy of an object. When the use of immutable objects is performed, cloning is essential. For example, for an immutable object that has a list field, the getter should always return a clone of the list to preserve immutability.

## How it’s implemented in Java (discuss & illustrate)

The clone() method will call the clone() method of its parent class to obtain a copy, until it reaches the uppermost class’ clone() method, which is the Object class. This creates a new occurrence of the class.

When using this method, the Cloneable interface needs to be implemented. If not, the program will throw an exception (CloneNotSupportedException).

## Options available for implementing cloning when cloning inheritance hierarchies

The implementation of the clone() method is by default a shallow copy, where a new object is created and the values from the original object are copied into the cloned object. This is also known as a field-by-field copy. Thus if the object cloned is a reference, the clone will be a reference too. The issue in such case is that both objects will then share the same reference. This means that any change to the original or cloned reference object will affect the other one.

The alternative to a shallow copy, is implementing a deep copy. For that fields are dereferenced. This means that objects referenced by the clone object are distinct to that of the original one. To do so using the clone() method, one has to first obtain a copy of the superclass, and then perform so within one’s clone() method.

## Alternatives to Java clone mechanism

The use of the clone() method can sometimes be tricky and can occur risks, if the implementation of the clone isn’t checked properly. That is we need to constantly keep track of what they are, shallow or deep clones, as they will have a different impact on the system. Thus, the use alternative methods, such as:

* **Factory method**

It is a creational pattern, and creates objects without needing to specify the exact class of the object (Factory method pattern, 2017). It does so by calling methods in an interface or implemented in a child class for example. This method relies on inheritance, as object creation is delegated to subclasses implementing the factory method.

* **Copy constructor**

A copy constructor has one parameter, which is the type of the class, and is used to create a copy of an existing object of the same class. It is also known as a conversion constructor (Constructor (object-oriented programming), 2017).

* **Serialization and deserialization**

Serialization is the process of converting an object into an array of bytes, so it can be saved on disk or sent through streams. Deserialization is the reverse process. For it to be implemented, it needs to implement the Serializable interface. This means that an object can be re-created, or an exact copy of it done.

# References

*clone (Java Method)*. (2017, 04 26). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Clone\_(Java\_method)

*Constructor (object-oriented programming)*. (2017, 4 26). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Constructor\_(object-oriented\_programming)#Copy\_constructors

*Factory method pattern*. (2017, 04 26). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Factory\_method\_pattern

# Appendix

## Task1 files

### Team.java

package task1;

public class Team {

private String teamName;

private int teamWin;

private int teamDraw;

private int teamLoss;

private int goalsFor;

private int goalsAgainst;

private int goalDifference;

private int teamPoints;

public Team(String name,int win, int draw, int loss, int gFor, int gAgainst, int points){

this.setTeamName(name);

this.updateTeamResult(win, draw, loss, gFor, gAgainst, points);

this.teamWin = win;;

this.teamDraw = draw;

this.teamLoss = loss;

this.goalsFor = gFor;

this.goalsAgainst = gAgainst;

this.goalDifference = gFor - gAgainst;

// this.teamPoints = points;

}

public void updateTeamResult(int win, int draw, int loss, int gFor, int gAgainst, int points){

teamWin += win;

teamDraw += draw;

teamLoss += loss;

goalsFor += gFor;

goalsAgainst += gAgainst;

goalDifference += (gFor - gAgainst);

}

// Constructors

public void setTeamName(String teamName) {

this.teamName = teamName;

}

public void setTeamWin(int teamWin) {

this.teamWin = teamWin;

}

public void setTeamDraw(int teamDraw) {

this.teamDraw += teamDraw;

}

public void setTeamLoss(int teamLoss) {

this.teamLoss += teamLoss;

}

public void setGoalsFor(int goalsFor) {

this.goalsFor += goalsFor;

}

public void setGoalsAgainst(int goalsAgainst) {

this.goalsAgainst += goalsAgainst;

}

public void setGoalDifference() {

goalDifference += (goalsFor - goalsAgainst);

}

// Getters

public String getTeamName() {

return teamName;

}

public int getTeamWin() {

return teamWin;

}

public int getTeamDraw() {

return teamDraw;

}

public int getTeamLoss() {

return teamLoss;

}

public int getGoalsFor() {

return goalsFor;

}

public int getGoalsAgainst() {

return goalsAgainst;

}

public int getGoalDifference() {

return goalDifference;

}

public int getTeamPoints() {

return teamPoints;

}

}

### Reflection.java

package task1;

import java.io.File;

import java.io.PrintStream;

import java.lang.reflect.\*;

import java.util.Arrays;

public class Reflection {

public static void main(String[] args) {

// // Obtain the class object if we know the name of the class

Class<Team> team = Team.class;

try {

PrintStream console = new PrintStream(new File ("reflectionFile.java"));

System.setOut(console);

// get the package name of the class

Package teamPackage = team.getPackage();

console.println(teamPackage);

console.println();

//class name

String teamClassNoPackage = team.getSimpleName();

console.println("public class "

+ teamClassNoPackage);

console.println();

Field[] privateField = Team.class.getDeclaredFields();

for (Field field : privateField) {

String name = field.getName();

Object type = field.getType();

console.println(type + " " + name);

}

// gets all the public member fields of the class Team

Field[] fields = team.getFields();

for (Field oneField : fields) {

// get public field name

Field field = team.getField(oneField.getName());

String fieldname = field.getName();

Object fieldType = field.getType();

console.println("public " + fieldType + " " + fieldname);

}

console.println();

// get all the constructors of the class

Constructor[] constructors = team.getConstructors();

for (Constructor constructor : constructors) {

console.println(constructor);

}

console.println();

// get all methods declared in the class

// but excludes inherited methods.

Method[] declaredMethods = team.getDeclaredMethods();

for (Method dmethod : declaredMethods) {

console.println(dmethod.getReturnType() + " " + dmethod.getName());

}

}

catch (Exception e) {

e.printStackTrace();

}

}

}

## Task 2 Files

### Race.java

package task2;

public class Race {

private int rank = 1;

public synchronized void getReadyToRace() throws InterruptedException{ //

this.wait(); // cars leave at the same time more or less...

}

public synchronized void startRace(){

this.notifyAll();

}

public synchronized int crossFinishLine(){ // rank value increases everytime a driver crosses finish line

return rank++;

}

}

### Main.java

package task2;

public class Main {

public static void main(String[] args) {

Race race = new Race();

Checkpoint check = new Checkpoint();

// Creation of new drivers

Thread t1 = new Thread(new Car("Sebastian Loeb", race, check));

Thread t2 = new Thread(new Car("Colin McRae", race, check));

Thread t3 = new Thread(new Car("Petter Solberg", race, check));

Thread t4 = new Thread(new Car("Sebastien Ogier", race, check));

Thread t5 = new Thread(new Car("Richard Burns", race, check));

try { // start of race

t1.start();

t2.start();

t3.start();

t4.start();

t5.start();

System.out.println("Cars line up...");

Thread.sleep(1000);

System.out.println("Ready...");

Thread.sleep(1000);

System.out.println("Set...");

Thread.sleep(1000);

System.out.println("GO!!!");

race.startRace(); // calls notifyAll() method

}

catch (InterruptedException e) {

e.printStackTrace();

}

}

}

### Checkpoint.java

package task2;

import java.util.Random;

public class Checkpoint { // this will geenrate a random number between 1 and 3 seconds for pit stop

private Random rdm = new Random(System.currentTimeMillis());

public synchronized long reFuel() throws InterruptedException{ // synchronized is used to ensure other cars wait their turn

long duration = Math.abs(this.rdm.nextLong() % 2000) +1000;

Thread.sleep(duration);

return duration;

}

}

### Car.java

package task2;

import java.util.Random;

public class Car implements Runnable {

private String name;

private Race race;

private Checkpoint check;

private Random rdm = new Random(System.currentTimeMillis());

public Car(String name, Race race, Checkpoint check) {

this.name = name;

this.race = race;

this.check = check;

}

public long raceLap() throws InterruptedException{ // how long it takes to do 1 lap

long duration = Math.abs(this.rdm.nextLong() %4000) +1000;

Thread.sleep(duration);

return duration;

}

public void run(){

try{

this.race.getReadyToRace();

System.out.println(this.name + " has crossed the start line");

for (int i = 1; i <=3; i++) {

long time = this.raceLap();

System.out.println("\n" + name + " has reached checkpoint " + i + " in " + (double)time/1000 + " minutes"); //put minutes to be more realistic of a race

time = this.check.reFuel();

System.out.println(name + " has refuelled for " + (double)time/1000 + " seconds and has now left checkpoint " + i);

System.out.println();

}

int place = this.race.crossFinishLine();

System.out.println(name + " finishes in position " + place);

}

catch (InterruptedException e) {

System.out.println("Oh no!");

}

}

}