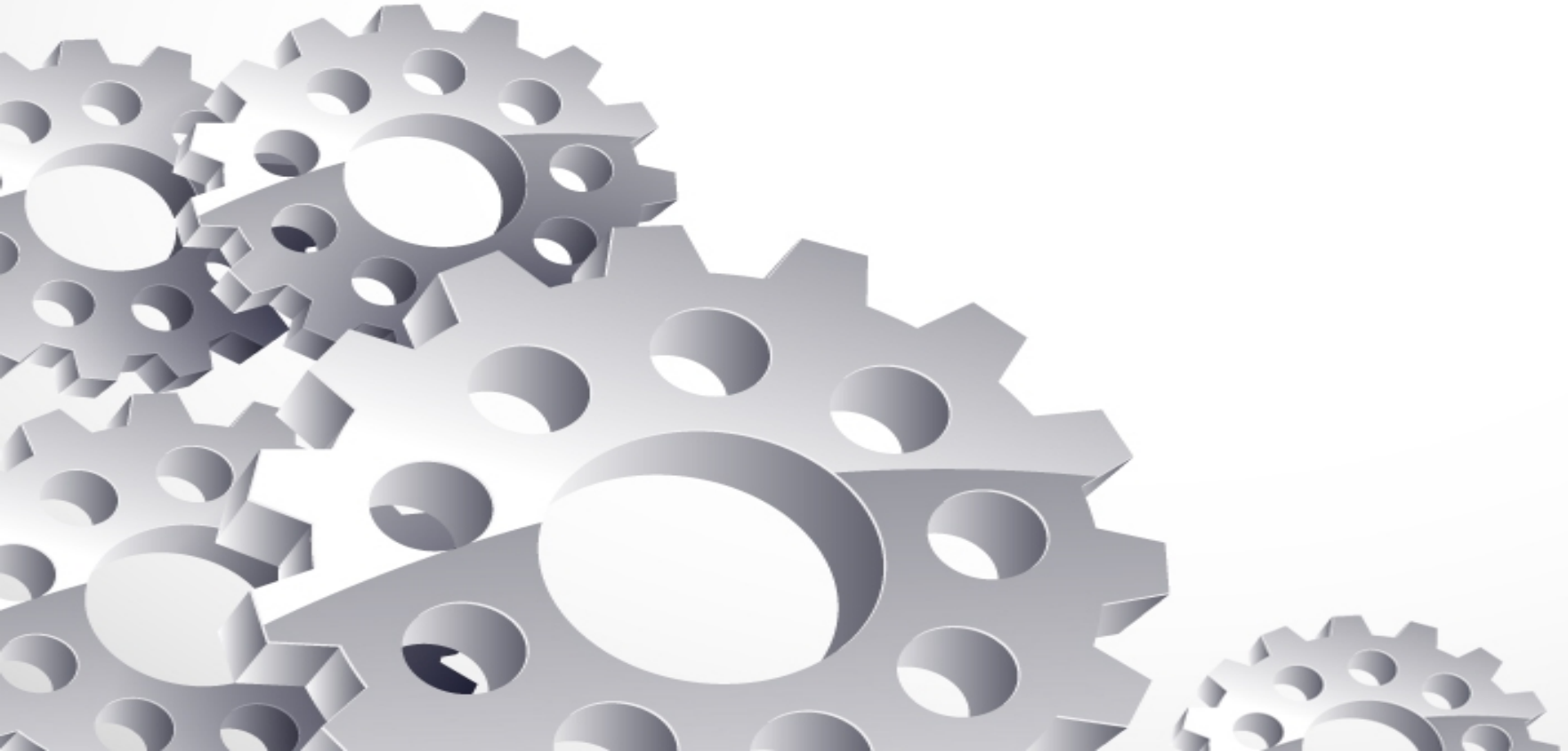
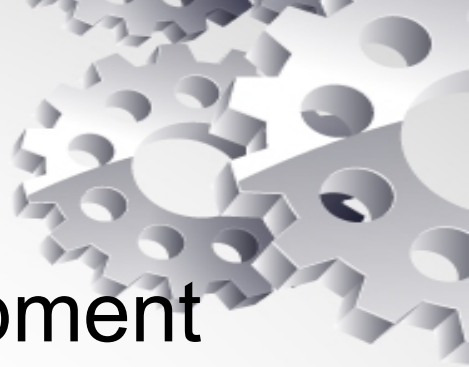
ADVANCED JAVA
(BEKİR)



GENERIC TYPES IN JAVA



- One of our responsibilities in the software development process is to deal with errors and try to solve them.
- The programming languages provides some feature us to prevent such errors.
- One of them is Generic Types

GENERIC TYPES IN JAVA

- Generics is a feature that enable us to work different reference data types.

- We can decide which reference type to work with.

GENERIC TYPES IN JAVA

- Before Generics (Before Java 1.4)

```
// Object
    List<Object> objectList = new ArrayList<Object>();
    objectList.add("Hello World");
    objectList.add(2022);
    String string  = (String) objectList.get(0);
    Integer integer = (Integer) objectList.get(1);
    // Raw
    List rawList = new ArrayList();
    rawList.add("Hello World");
    rawList.add(2022);
    String string  = (String) rawList.get(0);
    Integer integer = (Integer) rawList.get(1);
```

ClassCastException in Runtime

```
List list = new ArrayList();
list.add("Hello World");
list.add(2022);
for (int i = 0; i < list.size(); i++) {
    String string = (String) list.get(i);
}
```

GENERIC TYPES IN JAVA

- After Generics (With Java 5)

- New features ; Autoboxing, Unboxing, Foreach, static import, concurrency improvement, GENERICS etc.

- The language improved and changed with the coming of generics expressions. Many java libraries have been rewritten with Generics expressions.

GENERIC TYPES IN JAVA

Why Should We Use Generics

- Type-safety

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

```
it is required to specify the type of object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

- Type casting is not required

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

```
After Generics, we don't need to typecast the object.
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

- Compile-Time Checking

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

GENERIC TYPES IN JAVA

- Enabling programmers to implement generic algorithms
  - In many algorithms, logic remains same, but data type may change.
  - Generics make it possible to write classes, methods and interface that work with different kind of data in a type safe manner.

GENERIC TYPES IN JAVA

- Generic Types-Naming Convention
- Java Generic Type Naming convention helps us understanding code
- Uppercase letters to make it easily distinguishable from java variables

The most commonly used type parameter names:

- E-Element (extensively used by the Java Collection Framework)
- K-Key (Used in Map)
- N-Number
- T-Type
- V-Value (Used in Map)
- S, U, V - 2nd,3rd,4th types

GENERIC TYPES IN JAVA

- In Java 7, the compiler has been made smarter. When we create a generic instance, the necessity to specify the generic type again on the right has been removed.

- // Before Java 7
  - List<String> list = new ArrayList<String>();
- // After Java 7
  - List<String> list = new ArrayList<>();

GENERIC TYPES IN JAVA

- Generic Types
  - Generic Class or Interface

```
class name<T1, T2, ..., Tn> {
/* ... */
}

public interface name<E> {
}
```

  - Generic Method

These are some properties of generic methods:

- Generic methods have a type parameter (the diamond operator enclosing the type) before the return type of the method declaration.
- Type parameters can be bounded (we explain bounds later).
- Generic methods can have different type parameters separated by commas in the method signature.
- Method body for a generic method is just like a normal method.

```
public static <T> T identical(T obj){
        return returnValue;
 }
```

GENERIC TYPES IN JAVA

- Restricting the Generic Types
  - Bounded Type
    - Upper Bounded
      - » extends keyword
    - Lower Bounded
      - » super keyword
    - WildCard ?
      - ? extends Number
      - ? super Integer

GENERIC TYPES IN JAVA

– Type Erasure

Java compiler makes type erasure;

- With erasure, the bytecode after compilation contains only normal classes, interfaces and methods.
  - removes all type parameters and replaces them with their bounds
  - or with Object if the type parameter is unbounded
  - Insert type casts if necessary to preserve type safety.

GENERIC TYPES IN JAVA

– Type Erasure

```
public class GenType<T> {
  T obj;
  public T getObj() {
    return obj;
  }

  public void setObj(T obj) {
    this.obj = obj;
  }
}
```

```
public class GenType {
  Object obj;
  public Object getObj() {
    return obj;
  }

  public void setObj(Object obj) {
    this.obj = obj;
  }
}
```

```
public class GenType<T extends Number> {
  T obj;
  public T getObj() {
    return obj;
  }

  public void setObj(T obj) {
    this.obj = obj;
  }
}
```

```
public class GenType {
  Number obj;
  public Number getObj() {
    return obj;
  }

  public void setObj(Number obj) {
    this.obj = obj;
  }
}
```

GENERIC TYPES IN JAVA

REVIEW

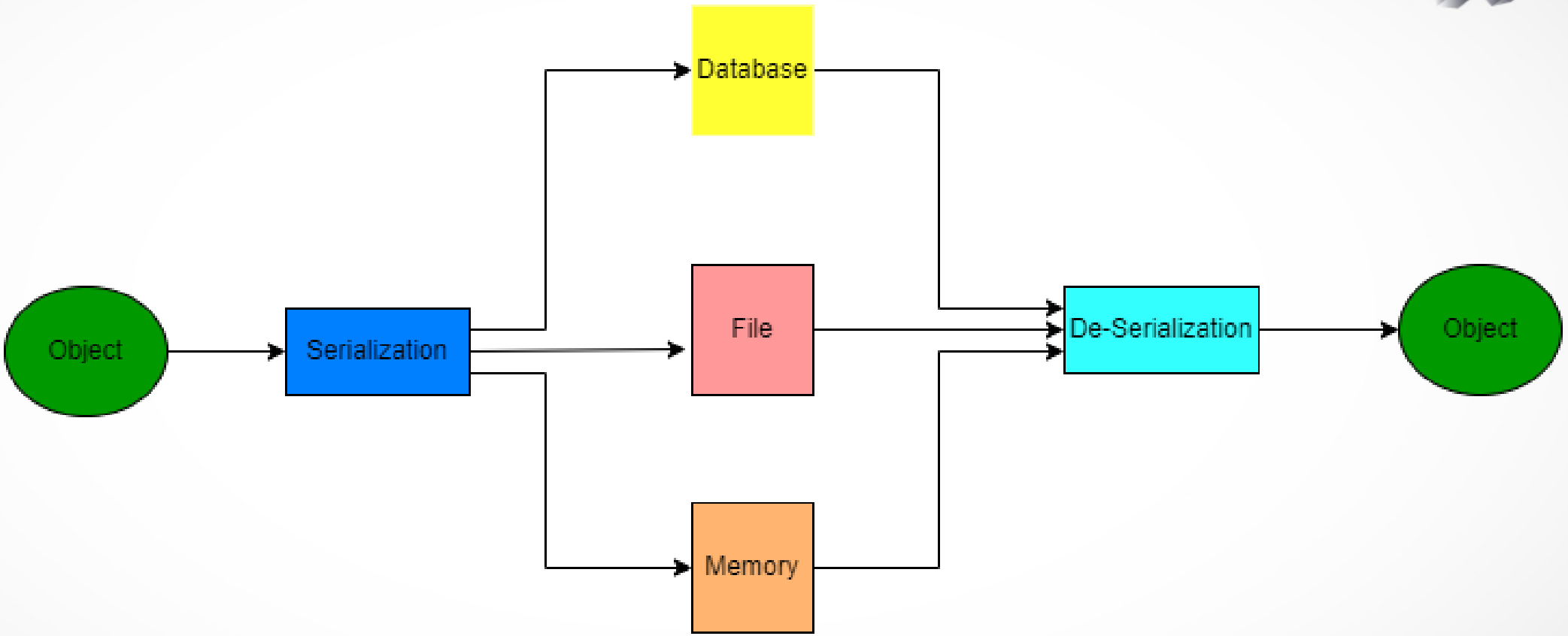
- Generics provide type safety as checking the type in compile time
- Generics don’t work with primitive types.
- You can have generic classes, generic methods and generic interfaces.
- A generic class with type parameters can’t have static members using the type parameter.
- Type parameters are erased when generic classes are compiled.

Serialization and De-Serialization in JAVA

- Serialization is a mechanism of converting the state of an object into a byte stream.

- Deserialization is the process of reconstructing a data structure or object from a series of bytes .It is the reverse process of the serialization. The byte stream is used to recreate the actual Java object in memory.

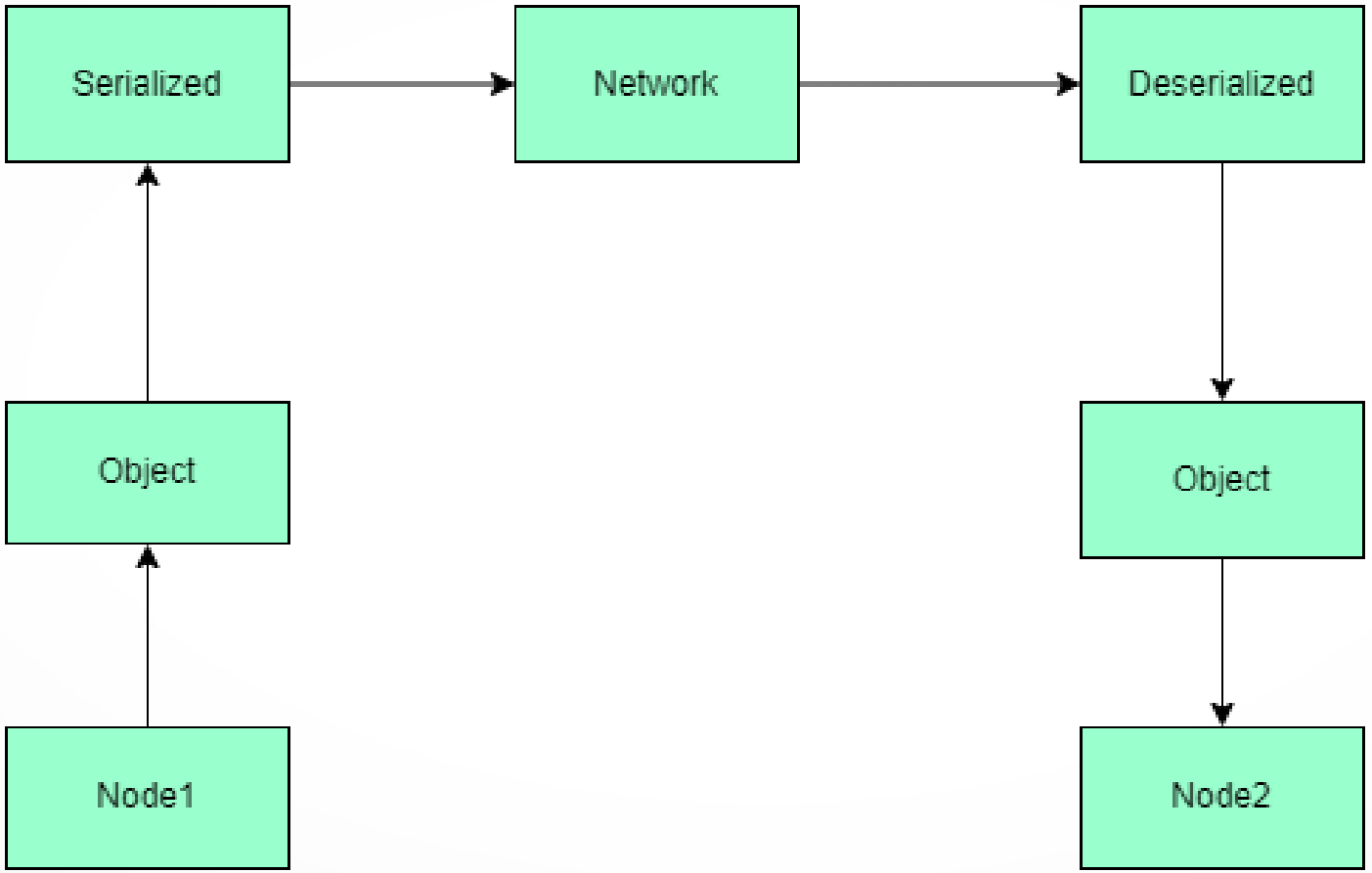
Serialization and De-Serialization in JAVA



Serialization and De-Serialization in JAVA

- Distributed systems share objects across separate nodes.
- Objects should be delivered over the wire.
- Serialization enables the programmer to save and transfer the object in a standardized way.
- Deserialization is the reverse process of the serialization. It enable the programmer to create objects after they were serialized for transmission over the wire.

Serialization and De-Serialization in JAVA



Serialization and De-Serialization in JAVA

```
class User implements Serializable{

private static final long serialVersionUID = 1L;

private Long id;
private String name;
private String ssn;

//Getter and Setter Methods

}
```

- The **Serializable** interface is present in java.io package.
- This Interface does not have any methods and fields.
- Classes that implements it do not have to implement any methods.
- Classes implement it if they want their instances to be Serialized or Deserialized
- serialVersionUID is a unique identifier for each class, JVM uses it to compare the versions of the class ensuring that the same class was used during Serialization is loaded during Deserialization.

Java Enums

- Java 5 first introduced the enum keyword
- It is a special type of class
- It represents a group of constants
- It makes the code more readable
- Allow for compile-time checking.

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

Java Enums

- All java enum implicitly extends java.lang.Enum class that extends Object class and implements Serializable and Comparable interfaces. So we can’t extend any class
- in enum.

- Enum is a keyword so we can’t end package name with it, for example com.tpe.enum is not a valid package name.

- we can safely compare them using “==” and equals() methods. Both will have the same result.

- Enum constructors are always private.

- We can’t create instance of enum using new operator.

- Enum constants are implicitly static and final

WHAT IS THE THREAD?

- Threads are independent, concurrent paths of execution through a program
- Each thread has its own stack, its own program counter, and its own local variables.
- Threads share memory, file handles, and other per-process state.
- Every Java program has at least one thread -- the main thread.

WHAT IS THE THREAD?

- When a Java program starts, the JVM creates the main thread and calls the program's main() method within that thread.
- Thread means a segment of a process
- Threads allows a program to operate more efficiently by doing multiple things at the same time

WHY USE THREADS?

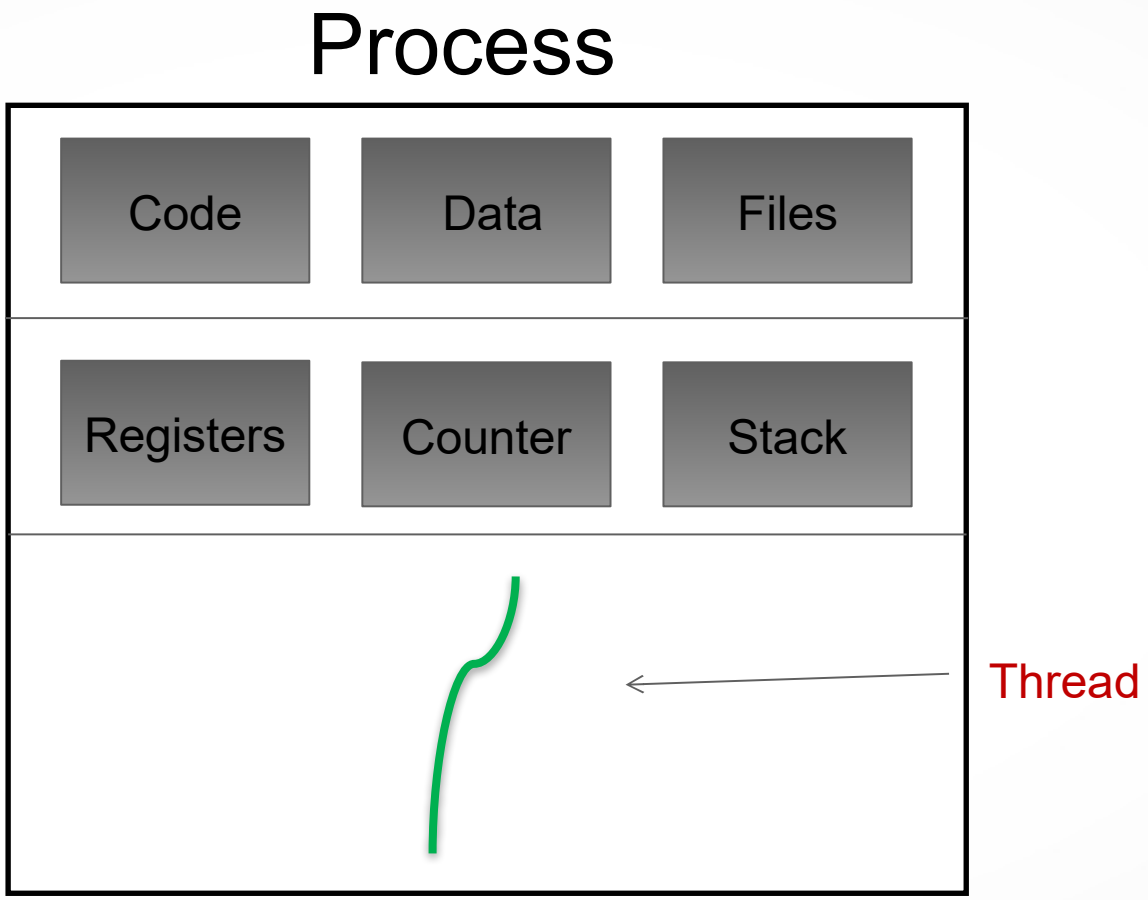
- There are many reasons to use threads in your Java programs. If you use Swing, servlets, RMI, or Enterprise JavaBeans (EJB) technology, you may already be using threads without realizing it.

- Some of the reasons for using threads are that they can help to:
  - Make the UI more responsive
  - Take advantage of multiprocessor systems
  - Perform asynchronous or background processing

WHAT IS MULTITHREADING IN JAVA

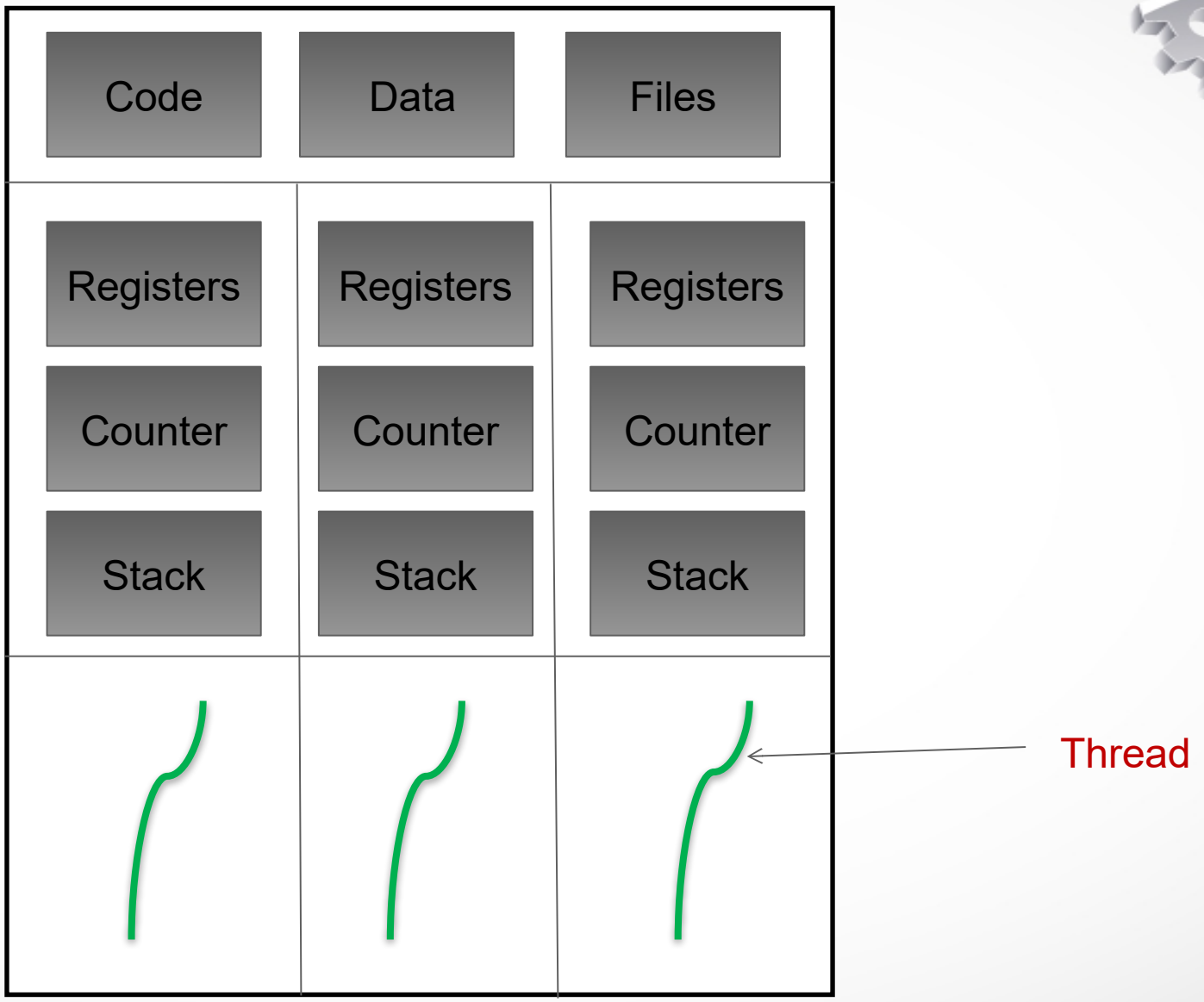
- Multithreading is a Java feature
- Multitreading allows concurrent execution of two or more parts of a program
- It is used to maximize utilization of CPU

MULTITHREADING IN JAVA



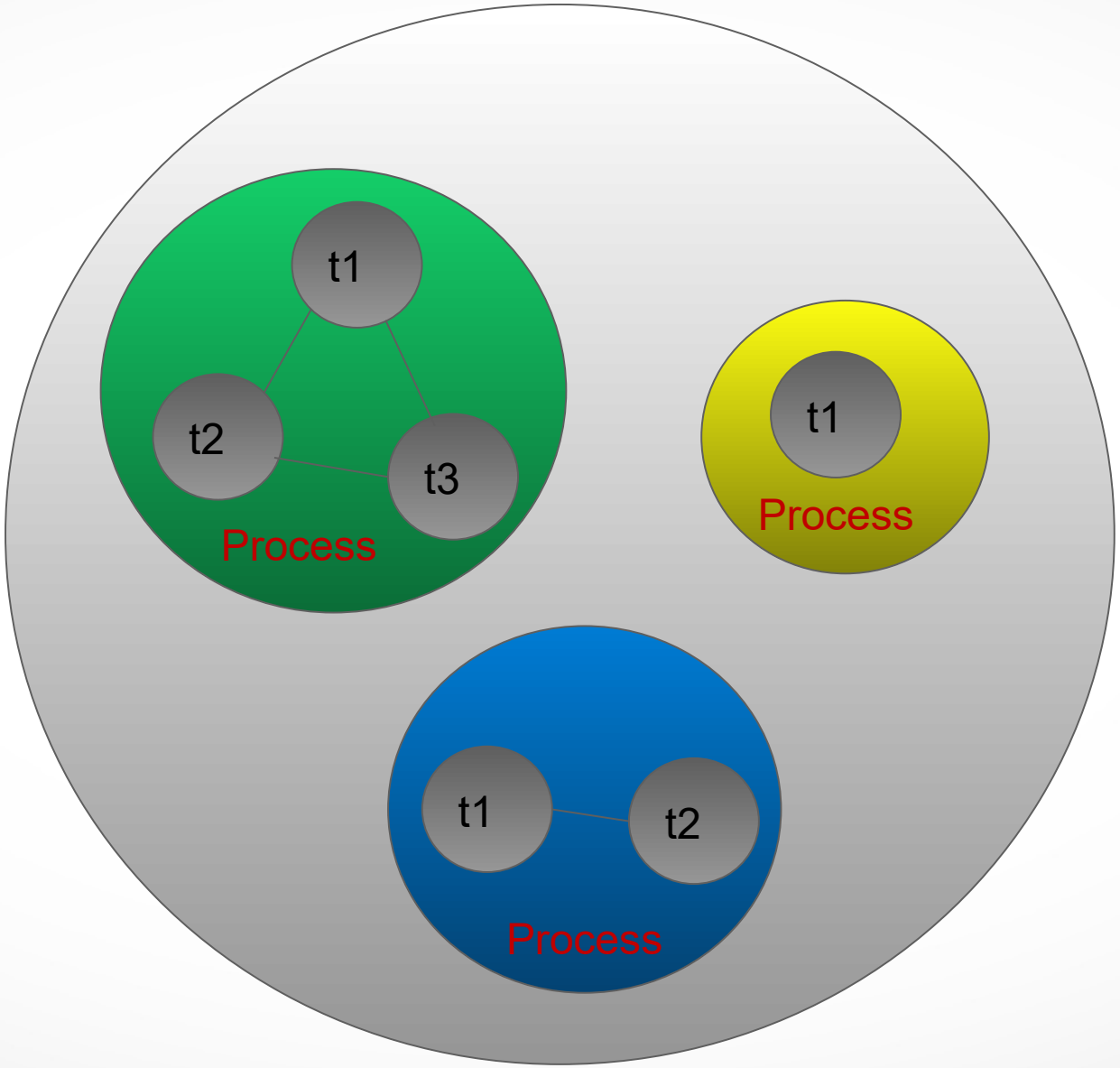
SingleThread
Process

MULTITHREADING IN JAVA



MultiThreaded Process

MULTITHREADING IN JAVA



CREATING THREAD IN JAVA

First way to create and start a thread- Extends Thread Class

```
class MyThread extends Thread{

@Override
public void run() {
System.out.println("It is custom thread");
  }
}

public static void main(String[] args) {

      MyThread myThread =new MyThread();
      myThread.setName("ExampleThread");
      myThread.start();
}
```

CREATING THREAD IN JAVA

Second way to create and start a thread- Implements Runnable Interface

```
class MyRunnable implements Runnable{

	@Override
	public void run() {
		System.out.println("Implemented Runnable interface");

	}
}

public static void main(String[] args) {

		Thread thread= new Thread(new MyRunnable());
		thread.start();

}
```

CREATING THREAD IN JAVA

First Way to create and start a thread

```
class MyThread extends Thread{

@Override
public void run() {
System.out.println("It is custom thread");
}
}

public static void main(String[] args) {

        MyThread myThread =new MyThread();
        myThread.setName("ExampleThread");
        myThread.start();

 }
}
```