Advanced Programming Methods

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

Java Reflection

- Introspection and reflection
- Java Reflection API

Concurrency

- Concurrency and parallelism
- Processes
- Threads
- Multithreaded programming in Java

Class loading in Java I

- A compiled Java program is described by a set of files having the **.class** extension. Each such file corresponds to a class of the program.
- These classes are not loaded into memory at application start. They load into memory only when they are first referenced: **lazy class loading**.
- Advantage: Lazy class loading can reduce memory usage.
- Loading is achieved by **class loaders**: these are objects that load Java classes during runtime dynamically to the JVM.

Class loading in Java II

- When a class is required by the JVM the class loader tries to locate the class (using the fully qualified name) - the bytecode .class file and load its definition into the runtime.

Class loading in Java III

- There are three types of class loaders:
  - **Bootstrap/Primordial ClassLoader**: starts the class loading process and loads system classes from the runtime jar (**rt.jar**, e.g. java.lang.*).
  - **Platform ClassLoader** (introduced in Java 9): responsible for loading the platform classes. Platform classes include Java SE platform APIs, their implementation classes, and JDK-specific run-time classes that are defined by the platform class loader or its ancestors.
  - **Extension ClassLoader**: loads the extensions of core java classes from the extension library. The extension directory mechanism was deprecated and removed starting from Java 9.

Class loading in Java IV

- **Application/System ClassLoader**: loads the application type classes.

- The Bootstrap class loader is written in C/C++, while all other class loaders are written in Java.

Class loading in Java V

- Every class loader has a *parent* class loader.
- The Java platform uses a delegation model for loading classes: when loading a class, the class loader delegates the request to the parent class loader and this happens recursively.
- If no parent class loaders find the class, then the child class loader will search for classes in the file system.
- If the class is still not found and the child class loader cannot load it, an exception will be thrown (java.lang.ClassNotFoundException).

Object lifecycle in Java I

1. The class is located on the disk and read into memory.
2. Java looks for static initializers that initialize static fields — fields that belong to the class (not to particular instances) and are shared by all objects created from the class.
3. When an object of a class is created or when a static field or method is accessed, the class is loaded.
4. When an object is created (with the new keyword), memory is allocated and a reference is created so that the runtime can keep track of the object.

Object lifecycle in Java II

5. The object is being used as long as necessary (allows access to public methods and fields).
6. When the object is no longer used it is removed from memory, the garbage collector destroys it.
7. When there are no more references to instances of a class the **class** object is marked for elimination from memory by the garbage collector.

Introspection and reflection

- The following slides are based on the documentation available at: Trail: The Reflection API.
- "Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine." (Java documentation)
- **Reflection** is the ability of a program to examine and modify the structure and behaviour (internal properties) of objects at runtime.
- Reflection is a feature in the Java programming language that allows an executing Java program to examine or "introspect" upon itself.

Advantages and uses of reflection I

- Instances of external classes (which do not belong to the application) can be created using their fully-qualified names $\rightarrow$ *extensibility*.
- Visual development environments use the information obtained by class browsers (which can enumerate members of a class) to help the developer.
- Debuggers must be able to access private data members of classes.
- Automated test frameworks must be able to call and use sets of set APIs defined on a class to test coverage.

Advantages and uses of reflection II

- Automated test frameworks (JUnit) use reflection to discover methods having the *@Test* annotation to call them when running the tests.
- Reflection can also be used to map the column names of a JDBC ResultSet or properties of a JSON file to getter/setter methods in a Java object.

Drawbacks of reflection

- If an operation is possible without reflection, it is preferable to do so, as operations using reflection have slower performance than their non-reflective counterparts because the JVM cannot perform certain optimisations on types that are dynamically resolved.
- Reflection cannot be used in restricted security environments, as it requires certain runtime permissions.
- Reflective code breaks abstractions (e.g. by performing operations that would be illegal in non-reflective code, such as accessing private fields and methods) and therefore may change behaviour with upgrades of the platform.

What can we do using reflection?

We can:

- Examine an object's class at runtime.
- Construct an object for a class at runtime.
- Examine a class' fields and methods at runtime.
- Invoke any method of an object at runtime.
- Change accessibility of constructors, methods and fields.

The Java **Class** object I

- For every type of object the JVM instantiates an instance of java.lang.Class.
- This instance can be used to:
  - inspect class members information and type information, at run-time;
  - create new objects of the class.
- It is also the entry point for all of the Reflection APIs.
- A class in a program is actually an object of the meta-class java.lang.Class

```
Class studentClass = Student.class;
```

The Java **Class** object II

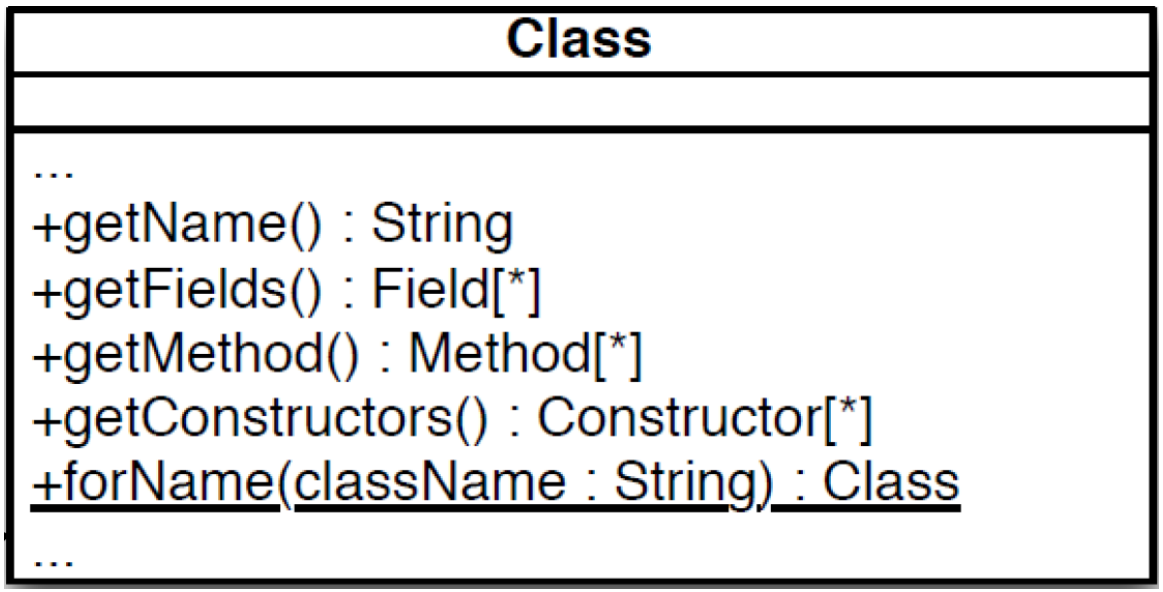
```
Student s = new Student(2, "Pop Andrei", 9.2);
Class studentClass2 = s.getClass();
```

The Java **Class** object III

- The class can also be retrieved using the static method Class.forName(), if the fully-qualified name of the class is available.

```
try {
    Class studentClass2 =
                Class.forName("reflection.Student");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

The Java **Class** object IV



Accessing package information

- Use the function Package getPackage().
- The package object contains information like:
  - the package name;
  - version information about the implementation and specification of a the package.
  - annotations.

```
Package p = studentClass.getPackage();
```

Accessing superclass or implemented interfaces information

- To get the superclass use the function Class getSuperclass().
- It returns a Class object, which can be used in the same manner - reflection can be applied on that as well.
- To get a list of the interfaces implemented by the class use the function Class[] getInterfaces().
- The function returns only the interfaces specifically declared.
- To get all the implemented interfaces, the process must be applied recursively.

```
Class superClass = studentClass.getSuperclass();
Class [] interfaces = studentClass2.getInterfaces();
```

Working with Constructors I

- To access the list of constructors use the function Constructor[] getConstructors().
- A Contructor object provides information about, and access to, a single constructor for a class.
- A specific constructor can be accessed if we know its exact parameters.

```
Constructor[] constructors = studentClass.getConstructors();

try {
    Constructor constructor =
        studentClass.getConstructor(String.class);
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```

Working with Constructors II

Getting constructor parameters

- Use the function Class[] getParameterTypes().

```
Class[] parameterTypes = constructor.getParameterTypes();
```

Instantiating objects using the Constructor object

- Use the method newInstance(), which takes a variable number of parameters.
- The number and types of parameters provided must be exactly the same as the number and types of parameters of the constructor.

Working with Constructors III

```
Constructor constructor = studentClass.getConstructor(
        int.class, String.class, double.class);
Student newObject =
    (Student) constructor.newInstance(2, "Marin Ioana", 9.4);
```

Working with Field objects I

- Use the function Field[] getFields() to get an array of all *public* fields of the class. To access private fields, the function getDeclaredFields() must be used.
- If the name of the public/private field is known, it can be accessed directly with the function getField(<field_name>)/ getDeclaredField(<field_name>);
- Only fields from the class are returned, not from superclasses too.
- Public field names and types can be retrieved with functions getName() and getType().
- Public fields can be get/set via an instance of the class owing the field. Private fields can be get/set too, but they must first be made accessible via the setAccessible(boolean) method.

Working with Field objects II

```
// private fields
Field[] fields = studentClass.getDeclaredFields();
try {
    // get field with a given name
    Field f = studentClass.getDeclaredField("average");
    // make the private field accessible
    f.setAccessible(true);
    Student st = new Student(2, "Marin Ioana", 9.4);
    // get value for fiels
    Object value = f.get(st);
    // set value for field
    f.set(st, 10);
    // now st.getAverage() will return 10
} catch (NoSuchFieldException e) { /*...*/}
catch (IllegalAccessException e) { /*...*/}
```

Working with methods I

- Use the function Method[] getMethods() to return all public methods of the class or interfaces and Method[] getDeclared-Methods to return all private methods of a class.
- Similarly to fields, we can get a certain public/private method by name and parameters with the function getMethod(String name, Class[] parameterTypes)/getDeclaredMethod(String name, Class[] parameterTypes).
- For a method we can get the parameter types (getParameter-Types()) and the return type (getReturnType()).
- A method can be called via invoke. This must receive the object on which it is invoked and the parameter values.

Working with methods II

```
Method[] methods = studentClass.getMethods();

try {
    // getting a method by name and parameters
    Method m = studentClass.getMethod("compareTo",
                                      Object.class);

    // getting the parameter types
    Class[] parameterTypes = m.getParameterTypes();

    // getting the return type
    Class returnType = m.getReturnType();
```

Working with methods III

```
        // invoking the method
        Student st1 = new Student(2, "Marin Ioana", 8.4);
        Student st2 = new Student(3, "Bojan Paul", 9.1);
        System.out.println(m.invoke(st1, st2));
    } catch (NoSuchMethodException e) { /*...*/ }
    catch (IllegalAccessException e) { /*...*/ }
    catch (InvocationTargetException e) { /*...*/ }
```

Working with annotations

- Annotations of a class can be retrieved with the function Annotation[] getAnnotations() applied on a Class object.
- Method annotations can be retrieved with the function Annotation[] getDeclaredAnnotations() applied on a Method object.
- Field annotations can be retrieved with the function Annotation[] getDeclaredAnnotations() applied on a Field object.

Example

Using reflection to generically read/write from/to CSV files

reflection.Main

Concurrency and parallelism I

- **Parallelism** refers to using multiple processors/cores to improve the performance of a computation.
- A system that is running in parallel can execute multiple operations *simultaneously*.
- **Concurrency** is concerned with managing access to a shared resource from different threads of the same process.
- In a concurrent environment operations can be potentially run simultaneously, but given that the resources are shared, operations might not be able to be performed that way.
- A computer system has many active processes and threads at once (even if there is only one execution core).

Concurrency and parallelism II

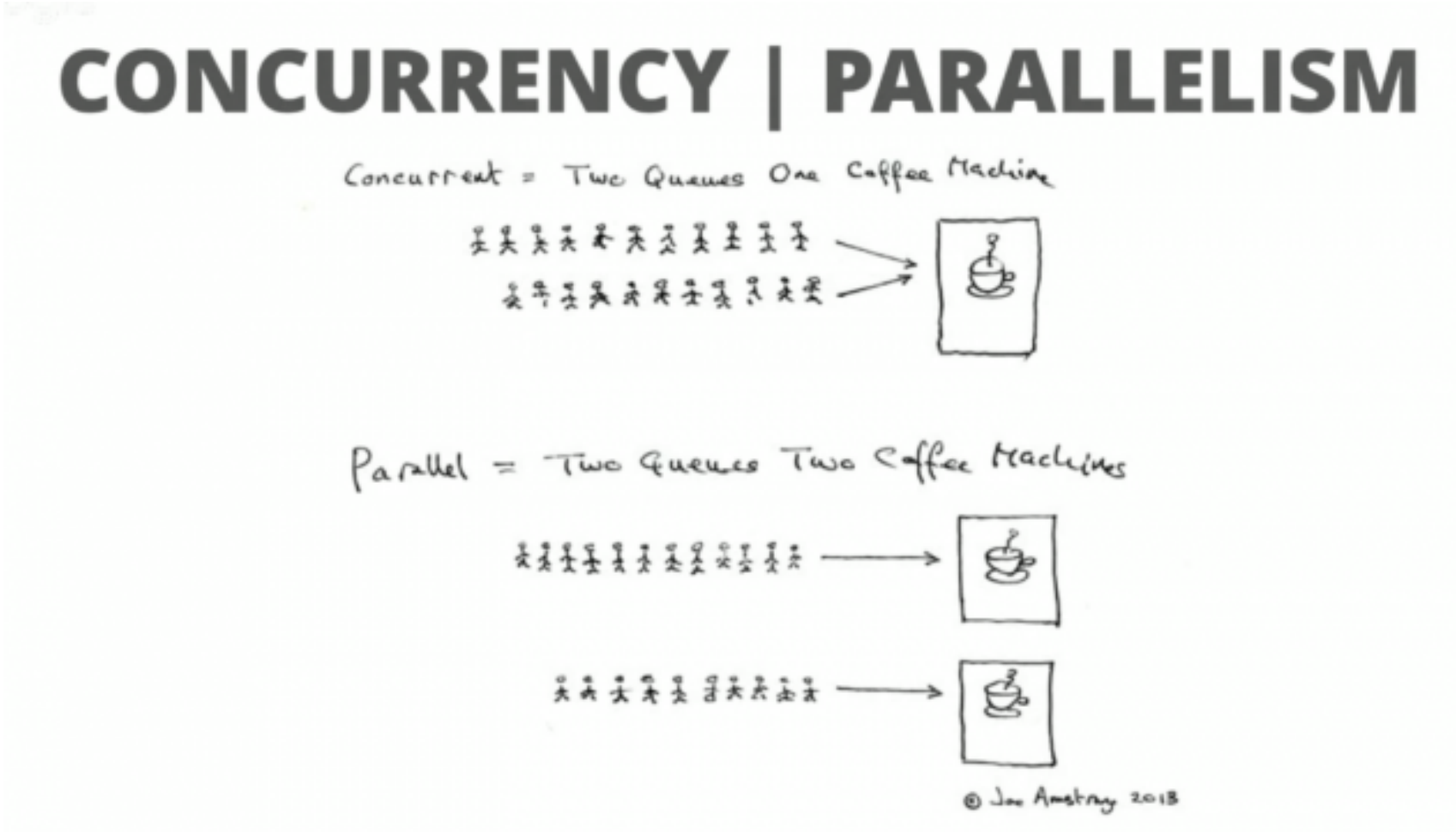


Figure: Figure source: Flavors of Concurrency in Java by Oleg Shelajev

Processes I

- A **process** is an instance of a program that is being executed.
- A process has its own execution environment and its own set of private resources (e.g. memory space, file handles).
- Processes are independent, isolated entities that are managed by the operating system.
- Processes running on the same machine or on different machines within a network can communicate between each other using Inter Process Communication (IPC) resources, e.g. *sockets* and *pipes*.

Processes II

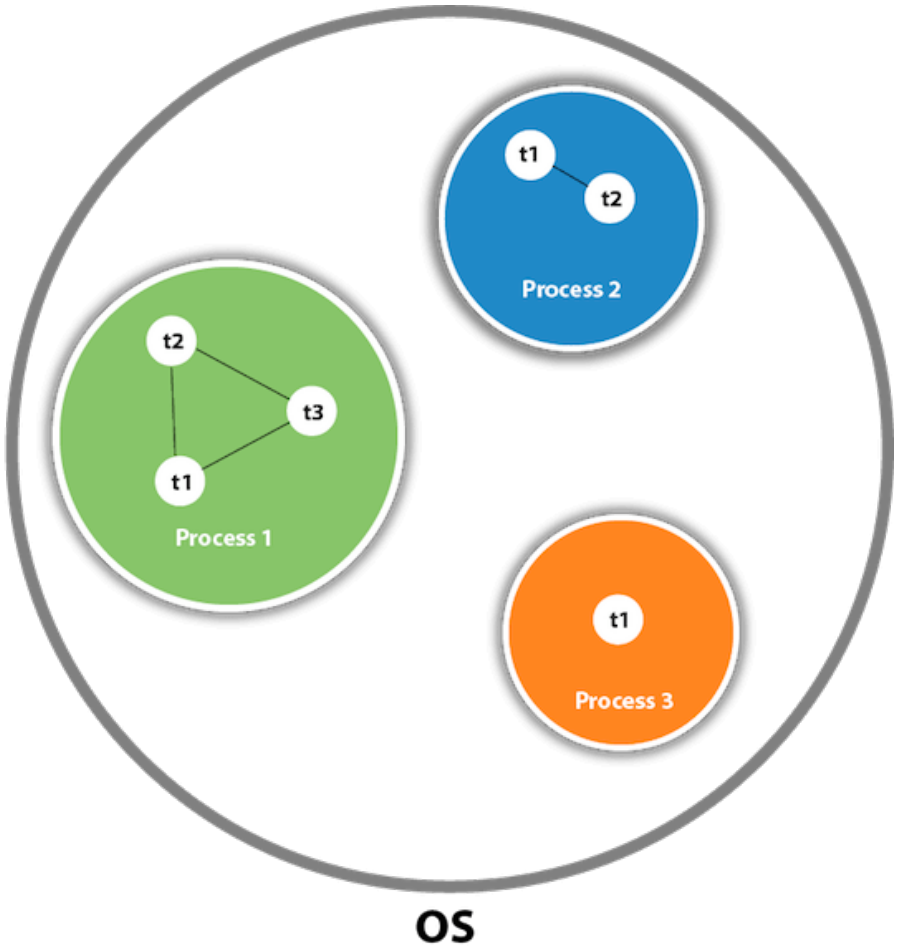


Figure: Figure source: Multithreading in Java

Threads I

- An application can execute more tasks at once.
- This supports a better use of the application's resources.
- Computationally expensive tasks should not block the main process (e.g. the GUI should not freeze when something computationally expensive happens).
- **Threads** are *lightweight processes* that are *apparently* executed in parallel within a process.
- Threads are not really executed simultaneously on a single core. The execution time on the CPU is divided into small time slices and the next time slice is assigned to the next waiting thread with the highest priority.

Threads II

- The switch happens really fast, such that threads are perceived to be running in parallel. Before switching, the operating system saves the state (context) of the currently running thread.
- This context switching adds extra overhead, therefore it is not as efficient as a true parallel execution.
- Threads share their resources (memory, file handles, etc.) with other threads of the same process.
- Creating a thread requires less resources than creating a process.
- Each process has at least one thread.

Threads III

- The threads within a process share the process resources (this leading to more efficiency), but communication may become problematic.
- All threads of a process have access to the heap.
- Given that Java allocates all objects on the heap, all threads have access to all objects.
- If one thread modifies an object, the modification is visible to the other threads.

Threads IV

- Each thread has its own stack, that cannot be accessed by other threads.
- The stack holds only local primitive variables and reference variables to objects in the heap.
- Stack memory is short-lived whereas heap memory lives from the start until the end of application execution.
- After all method calls are completed, the stack becomes empty.

Advantages of multithreaded programming

- Enhanced performance: better use of system resources, time saving.
- Better responsiveness (e.g. GUI).
- Economy: allocating memory and resources is expensive operations in terms of computational time and space. Threads share such resources.
- In multi-core and multi-processor systems threads can be run in parallel, each one solving a smaller task. If an application is multi-threaded, it is therefore easily scalable to multi-processor architectures.

Costs of multithreaded programming

- The program might become overheaded, increasing its complexity.
- Creating and destroying threads involves computational resources.
- Sharing resources must be correctly managed, which also increases program complexity.
- Debugging is difficult in a multithreaded application. It is also difficult to reproduce the same results.

The Java Object class - methods for multithreading

- The Object class offers some methods that are used in multi-threaded programming:
  - wait() - forces the current thread to wait, until another thread gives some notification (via the notify() or notifyAll() method).
  - notify() - allows notifying other objects when a certain event occurs. This function wakes up a single thread that is waiting on this object's monitor.
  - notifyAll() - allows notifying more objects when certain events occur. This function Wakes up all threads that are waiting on this object's monitor.
- But... what is a monitor?

What is a monitor?

- A monitor is a synchronisation mechanism, allowing to control concurrent access to some section (designed in the early 1970s, it is general, not limited to the Java language).
- A monitor provides the following main capabilities:
  1. *mutual exclusion*: it ensures that only one thread at a time has mutually exclusive access to a critical section.
  2. *cooperation*:
     - threads can wait (they are blocked) for certain conditions to be met;
     - threads can notify each other when some conditions they have been waiting for are met.
- Java provides built-in support for monitor objects - we will get to that later on.

The Java Thread class I

```
public class Thread extends Object implements Runnable
```

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- The main methods of the class Thread are as follows:
  - start() - launches the new thread. After a new thread is launched, the program execution is controlled by at least two threads: the current thread and the new thread.
  - run() - defines what the new thread should do. The thread's activity is described in this method.
  - sleep() - causes the thread to sleep (temporarily cease execution).

The Java Thread class II

- join() - causes the calling thread to pause its execution until the current thread terminates.
- yield() - A hint to the scheduler that the current thread is willing to yield its current use of a processor, to allow another thread to run.
- isAlive() - tests if the thread is alive.
- setDaemon(boolean) - marks the thread as a daemon thread or a user thread. A **daemon thread** is a low-priority thread which provides services to user threads. Usually daemon threads' run() method contains an infinite cycle. If all remaining threads are daemon threads, the JVM will not wait for them to finish before terminating. The garbage collector is an example of a daemon thread.
- getPriority(), setPriority(int) - gets or sets the priority of the thread.

Creating new threads in Java

- Any running Java program contains a thread.
- This thread is not explicitly created by the programmer, it is automatically created by the JVM.
- This thread automatically calls the main method.
- We can explicitly create threads by:
  - Implementing the Runnable interface - recommended (inherit only if you want to override some behaviour).
  - Extending the Thread class.

Threads - example

threads.Main

Summary I

- Java uses lazy class loading.
- Reflection is the ability of a program to examine and modifythe structure and behaviour of objects and runtime.
- With reflection we can:
  - Examine an object's class at runtime.
  - Construct an object for a class at runtime.
  - Examine a class's field and method at runtime.
  - Invoke any method of an object at runtime.
  - Change accessibility to constructor, methods and fields.

Summary II

- **Parallelism**: a system that is running in parallel can execute multiple operations simultaneously.
- **Concurrency**: operations can be (apparently) run simultaneously and resources are shared.
- **Processes** are independent, isolated entities that are managed by the operating system.
- **Threads** are *lightweight processes* that can be executed in parallel within a process.
- Java offers support for multithreaded programming.
- *Next week:*
  - Multithreaded programming (cont.).
  - Design patterns.