Object Oriented Programming (part 2)

Agenda

- Generics
- The Java Platform Module System (JPMS)

Generics

What are generics ?

- Generics are introduces in JDK 1.5
- Provide the possibility to create parametrized types (classes/interfaces) and methods
- The parameters specify particular types that are bound to the target type or method

What are generics ?

- Provide compile time type safety
- Effetively this elliminates the need to create multiple similar classes

```
public class Table<D> {
    D details;

    public D getDetails() {
        return details;
    }
}
```

Generics vs C++ templates

- The equivalent of generics in the C++ world are templates
- However unlike Java generics C++ templates preserve type information at runtime !
- There is a similar effort undergoing for Java under project Valhalla
- Using generics requires use of type casting in many scenarios for that reason

Generics and collections

- A heavy user of Java generics are the standard Java collections

```
public interface List <E> {
    void add(E x);
    Iterator<E> iterator();
}
```

```
List<String> users = new ArrayList<String>();
users.add("Tom");
users.add("Jerry");
```

We are going to cover Java collections extensively in the next session !

Multiple generic types

- Multiple generic types can be defined (separated with commas)

```
public interface Map<K,V> {

    K getKey();

    V getValue();
}
```

Bounded types

- Generic types can be bounded with the **extends** keyword

```
public class Table<D extends Details> {
    D details;

    public D getDetails() {
        return details;
    }
}
```

Multiple bounds

- Multiple bounds can be specified with &
- If one of the bounds is a class it must be specified first

```
public class Table<D extends Details & Cloneable> {
    D details;

    public D getDetails() {
        return details;
    }
}
```

Generic type equivalence

- Types that use different generic parameters are neither equivalent nor subtypes !

```
List<Object> items = new LinkedList<Object>();
List<String> names = items; // does not compile
       // although String is a subclass of Object !
```

Wildcards

- In many cases you don't care at compile time about the generic parameter

- In that case a special wildcard (?) symbol can be used

```
public void printTableDetails(Table<?> table)
{
        ...
}
```

Wildcards

- Since wildcards denote an unknown element and are used for reference it is not permitted to perform operations that require the type (such as new Object creation)

- The following is not allowed:

```
List<?> items = new LinkedList<Object>();
items.add(new Object()); // compilation error
```

Bounded wildcards

- Wildcards can further limit the classes that can be passed

- We can specify a class that the wildcard type must extend which bounds the wildcard to all subclasses of that class

```
public void printTableDetails(
        Table<? extends Details> table) {
        ...
}
```

```
Table<TableDetails> table = new Table<TableDetails>();
printTableDetails(table) // at compile time the compiler
        // checks that TableDetails extends Details
```

Bounded wildcards

- Wildcards can also have a lower boundary using the **super** keyword

- This means that the generic type must be a parent class of the specified type

```
public void printTableDetails(
        Table<? super TableDetails> table) {
        ...
}
```

```
Table<Details> table = new Table<Details>();
printTableDetails(table) // at compile time the compiler
        // checks that Details is a parent class of
        // TableDetails
```

Is this allowed ?

```
public void addDetails(
       Table<? extends Details> table) {
    table.setDetails(new TableDetails());
       // signature: setDetails(D details);
}
```

Hint: check a few slides back about semantics of generic wildcards

Generic methods

- Methods in Java can also use generic types similar to classes

- This is especially useful in eliminating the need to have multiple overloaded methods with similar structure

```
public <Det> void printDetails(Det details) {
    if(details instanceof TableDetails) {
        System.out.println(((TableDetails)
                details).getColor());
    }
}
```

Generic methods

- The generic type can also be inferred when calling the method

```
Collections.sort(
        Collections.<String>emptyList());
```

Nested generics

- Generic types can also be used as types of another generic types (i.e. be nested)

```
Table<List<Details>> tableWithMultipleDetails =
        new Table<>();
```

Note that the <> syntax, also called **diamond operator** was introduced in Java SE 7 as way to minimize the amount of code written as the generic type can be deducted from the reference variable

Generics and legacy code

- In order to provide interoperability with legacy code specifying generic types might be omited
- Reference types without specifying generic types are also called 'raw types'
- Raw types are similar to wildcard types with lesser compile-time checks

Raw generic types

```
List<String> typedItems = new LinkedList<>();
List items = typedItems;
       // valid but generates warning in the IDE
items.add(new Object());
       // this is also valid !
       // (if items was a wildcard it would have
       // been a compiler error)
```

Is this allowed ?

```
public <Det> void printDetails(Det details,
                Object item) {
        if(details instanceof Det) {
                ...
        }
}
```

Generics and **instanceof**

- **instanceof** checks CANNOT be performed with generic types

- Wildcards and rawtypes however can be passed to the instanceof operator

```
t instanceof Table // valid
t instanceof Table<?> // valid
t instanceof Table<Details> // invalid !
```

Generics and casts

- Casts can be performed with generic types

```
Table<Details> table = (Table<Details>) getTable()
       // valid but generates warning
```

Generic restrictions

- Additional restrictions on generics we haven't covered so far include:
  - The type of a generic type used as a method parameter cannot be used to create instances
  - The type of a generic type cannot be used for static fields
  - Arrays of generic types cannot be created
  - Generic exceptions cannot be created, caught or thrown
  - Cannot overload a method with another one whereby once generic types are erased the two methods have the same name and list of parameters

Project Valhalla

- There is an undergoing effort conducted under project Valhalla to extend generics with the following:
  - **generic specialization**: the ability to pass primitive types as generic types (i.e. List<int>)
  - **reified generics**: the ability to retain generic type information at runtime

The Java Platform Module System (JPMS)

Modularity 101

- Standard Java libraries are modules - Hibernate, log4j and any library you can basically think of …

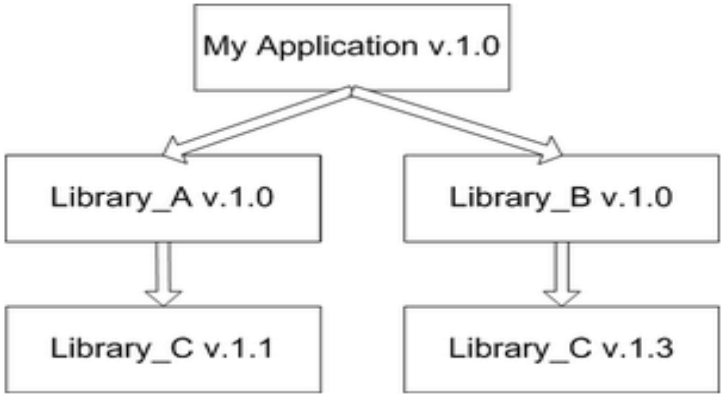
- Build systems like Maven provide transparent management of modules

Modularity 101

- Benefits of modularization:
  - smaller modules are typically tested easier than a monolithic application
  - allows for easier evolution of the system - modules evolve independently
  - development of the system can be split easier between teams/developers
  - increased maintainability of separate modules

Modularity 101

- The dependency mechanism used by the JDK introduces a number of problems that modular systems aim to solve:
  - The "JAR hell" problem caused by shortcomings of the classloading process



Modularity 101

- The dependency mechanism used by the JDK introduces a number of problems that modular systems aim to solve:
  - The lack of dynamicity in managing dependent modules
  - The lack of loose coupling between modules

Modularity 101

- Module systems aim to solve the mentioned problems and typically provide:
    - module management
    - module deployment
    - versioning
    - dependency management
    - module repositories
    - configuration management

OSGi

- Before the introduction of built-in modules in JDK 9 the de-facto standard module system in Java was OSGi
- OSGi is a set of specifications (core and compendium) that define a module runtime implemented in Java
- The module system introduced in JDK 9 is NOT a replacement of OSGi

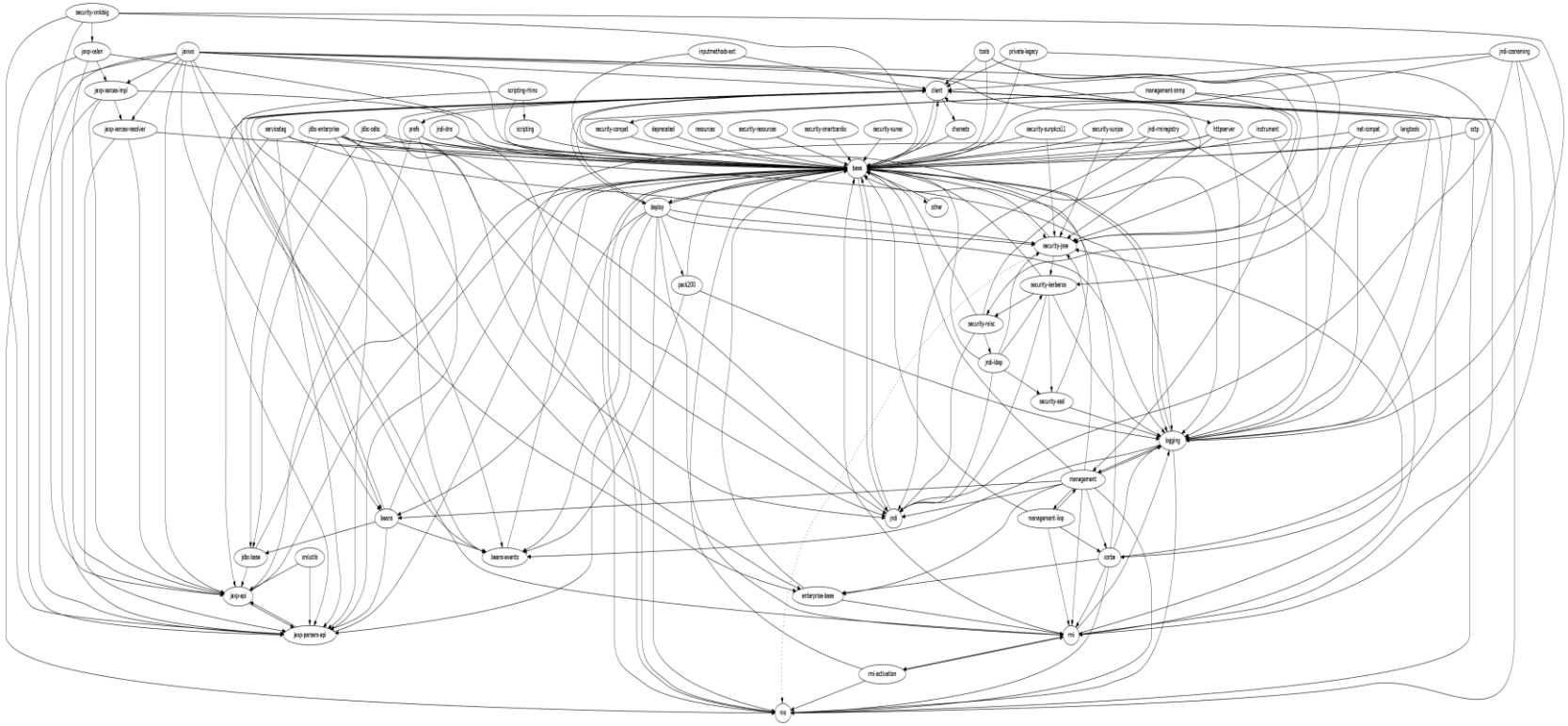
JDK modularity

- When speaking of modularity we should also consider the entire runtime (rt.jar) and the JDK core libraries …

- … and built-in support for improved "OSGi-like" modules in the Java platform

JDK modularity

- The JDK is monolithic …



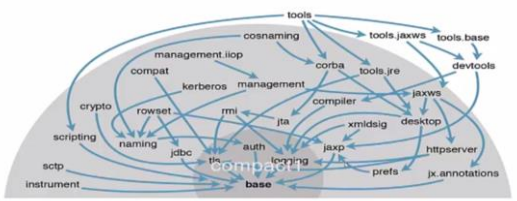
JDK modularity

- Some preliminary work such as compact profiles and removed/deprecated library dependencies is already done in JDK 8 …

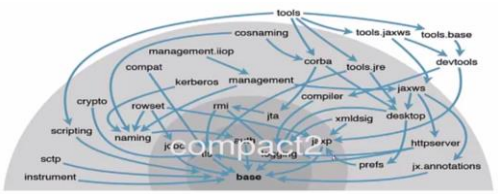
- **Compact profiles** provide smaller versions of the JDK

```
javac -profile <profile_name>
```

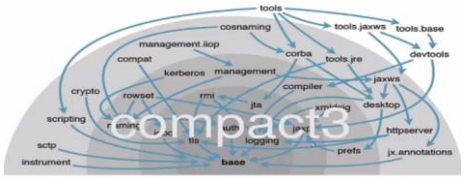
compact 1



compact 2



compact 3



Java Platform module system

- The Java Platform module system is a built-in module system introduced in JDK 9

- Splits the JDK classes into modules that can be listed with the following:

```
java –list-modules
```

Modularization of the Java platform is a significant change that impacts the entire Java ecosystem. Since it is not backward compatible with older versions some existing projects require significant effort to get them migrated to JDK 9 or newer.

Java Platform module system

- Introduces the following significant changes to the JDK:
  - definition of JDK internal modules
  - reorganization of JDK sources into modules and adoption of the JDK build system to build JDK modules
  - restructuring of the JDK build images to include modules
  - encapsulation of internal APIs (such as sun.misc.Unsafe)

Java Platform module system

- Introduces the following significant changes to the JDK:
  - Introduction of the jlink tool used to build a custom JDK image
  - new syntax for the Java language used to describe modules in **module-info.java** file

Java modules

- A Java module is describe in a **module-info.java** file

```
module com.example.services {
    requires com.example.entities;
}
```

- A module can be packaged in any of the following formats:
  - JAR file format
  - JMOD file format
  - JIMAGE file format

The java module descriptor

- The **module-info.java** file may contain the following:
  - Name of the module
  - Module dependencies
  - Public packages
  - Provided services
  - Consumed services
- All packages that are not exported in the module descriptor and private for the module

Module types

- The following types of modules are identified by the Java module system:
  - system: the modules of the JDK itself
  - application: the modules of a Java application
  - automatic: the modules corresponding to the Java libraries put on the module path
  - unnamed: A single module that contains all the classes from the classpath of the application
- The module path is provided with **--module-path** parameter for the JVM and is the equivalent of the classpath but for modules

Module dependencies

- Dependencies on other modules that must be provided on the module path is specified with **requires** elements

```
module com.example.services {
    requires com.example.entities;
    requires com.example.othermodule;
}
```

- The module dependencies must be located both at compile and at runtime

- The **requires static** element can be used to make the dependency optional at runtime

Module dependencies

- Dependencies on other modules can also be transitive

```
module com.example.services {
    requires transitive com.example.entities;
}
```

- This means that any modules that require the primary one will not have to require its transitive dependencies with **requires** elements

There is an implicit **java.base** module that does not need to be required explicitly by other modules

Module packages

- The packages that other modules can use from a target module need to be specifies explicitly with an **exports** element

```
module com.example.services {
    exports com.example.services.users;
}
```

- We can restrict exported package to certain modules using **exports ... to**

```
module com.example.entities {
    exports com.example.entities.users
       to com.example.services
}
```

Module services

- A module may declare that it uses a service
- The service might be implemented by some other module

```
module com.example.services {
    uses com.example.shared.services.Service;
}
```

- The service can then be retrieved for use via the java.util.ServiceLoader utility

```
ServiceLoader.load(
      com.example.shared.services.Service.class)
```

Module services

- A module may provide a service implementation for use by other modules

```
module com.example.shared {
    provides com.example.shared.services.Service
       with com.example.shared.services.ServiceImpl
}
```

Reflective access

- A module may specify explicitly that other modules may perform reflection on the members of a certain package

```
module com.example.shared {
    opens com.example.shared.services
}
```

- Reflective access can be restricted to certain modules

```
module com.example.shared {
    opens com.example.shared.services
       to com.example.services
}
```

We are going to cover reflection in the sessions related to the JDK APIs

Migrating to Java 9

- It is not necessary to migrate all of your projects to Java modules in order to migrate to JDK 9 (or newer)
- The classpath entries are put in an unnamed module
- However running without any extra changes required might not be the case for many projects for a number of reason (i.e. usage of internal APIs)

Migrating to Java 9

- Migration steps at a glance:
  - Install JDK 9 (or newer)
  - Try running the project
  - Run **jdeps** to analyze which JDK packages and internal APIs does the application use
  - Refactor your application accordingly so it compiles and runs fine under JDK 9 (or higher)

Migrating to Java modules

- Once a project is migrated to JDK 9 it can also be modularized which is typically of lesser importance

- The main intent of the Java platform module system is to split the monolith JDK into smaller modules rather than provide application developers a full blown module system

Migrating to Java modules

- To modularize a project:
  - proper **module-info.java** descriptors need to be added with proper declarations to the project
  - Third-party libraries might need to be upgrade accordingly or moved from the classpath to the modulepath
  - the module system will treat the libraries on the modulepath without **module-info.java** file as automatic modules

Split packages

- The module system implies a restriction on packages of the application's modules
- It is not allowed to have the same package exported from different modules
- This is the so called **split packages** problem
- There are different ways to deal with the problem depending on the scenario

Migration challenges

- The following issues might occur during migration to JDK 9 or later:
  - encapsulated internal JDK APIs
  - unresolved modules
  - split packages
  - cyclic dependencies
  - resolution of automatic module versions
  - removed JDK methods
  - removed rt.jar, tools.jar and dt.jar

Questions ?