Core JDK APIs

Agenda

- XML and JSON processing
- Date and Time
- Locales

Agenda

- Security
- GUI programming

Overview

Java SE

- Apart from the Core language Java SE (standard edition) comes with a number of useful utilities …

collections	Io/nio	date & time
object comparison	object cloning	XML/JSON parsing
concurrency	object serialization	security
networking	logging	GUI APIs
JDBC	process API	JShell
JMX	reflection	
lambdas	regex	
streams	locales	

In blue are the ones already covered and in green are the ones for which there is a dedicated session

XML and JSON processing

Parsers

- Parsing libraries for a variety of formats (not only XML and JSON) can be used

- They can be categorized as follows:

  - low level parsers where the application needs to convert the parsed format (i.e. XML or JSON) into a proper data structure

  - high level parsers that map directly the parsed format into Java objects of a particular class or set of classes

Low level XML parsing

- Low-level parsing capabilities are provided by the JAXP (Java API for XML Parsing) utilities in the Java platform
- JAXP provides the following types of XML parsers:
  - SAX parser: event-driven element-by-element processing
  - DOM parser: reads the entire model of the parsed XML in memory
  - StAX parser: even-driven parsing API with simpler model than SAX-based parsers and less memory consumtion than DOM-based parsers

SAX parser

```
public class XMLHandler extends DefaultHandler {

	@Override
	public void startElement(String uri,
		String localName, String qName,
		Attributes attributes) throws SAXException {
	}
	@Override
	public void endElement(String uri,
		String localName,
		String qName) throws SAXException {
	}
	public void startDocument() throws SAXException {
	}
	public void endDocument() throws SAXException {
	}
}
```

A handlers needs to be defined to handle XML parsing events

SAX parser

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setNamespaceAware(true);
SAXParser saxParser = spf.newSAXParser();
XMLReader xmlReader = saxParser.getXMLReader();
xmlReader.setContentHandler(new XMLHandler());
try (FileInputStream fis =
       new FileInputStream("table.xml")) {
       xmlReader.parse(new InputSource(fis));
} catch (FileNotFoundException e) {
       e.printStackTrace();
} catch (IOException e) {
       e.printStackTrace();
}
```

table.xml

```
<table serialId="123">
       <color>green</color>
       <size>100</size>
       <price>200</price>
</table>
```

DOM parser

```
DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new File("table.xml"));
System.out.println(doc.getFirstChild()
      .getNodeName());
NodeList children = doc.getFirstChild().getChildNodes();
for(int i = 0; i < children.getLength(); i++) {
      System.out.println(children.item(i)
            .getNodeName());
      System.out.println(children.item(i)
            .getTextContent());
}
```

StAX parser

```
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLEventReader eventReader =
factory.createXMLEventReader(new FileReader("table.xml"));

while (eventReader.hasNext()) {
        XMLEvent event = eventReader.nextEvent();
        switch (event.getEventType()) {
                case XMLStreamConstants.START_ELEMENT: {
                                System.out.println(
                                event.asStartElement()
                                .getName());
                }
        }
}
```

StAX parsers work in streaming fashion as SAX parsers but work in **pull** manner instead of **push** as SAX (where events are pushed to a handler)

JAXP capabilities

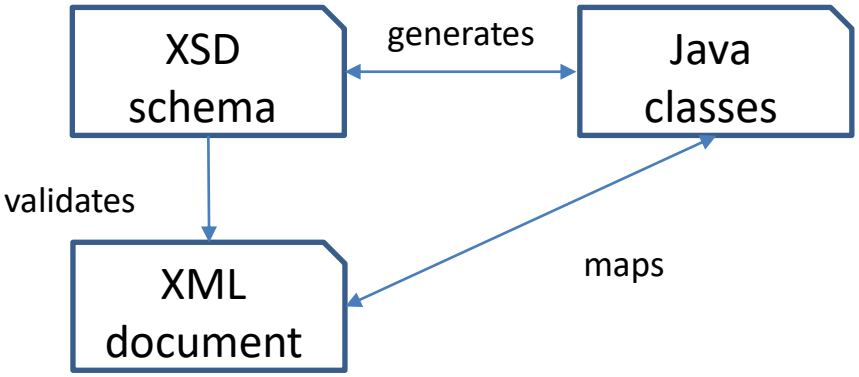
- The JAXP API provides a number of additional capabilities such as:
  - validation of the XML documents using DTDs or XSD schemas
  - error handling capabilities during XML parsing
  - XML traversal capabilities through XPath (which provides path-based domain-specific language for navigation in a XML document)
  - XML document transformations using XSLT

High level XML parsing

- The JAXB API is a higher level API than JAXP and provides the possibility to parse XML documents and map them directly to Java objects

- JAXB classes can be generated from an XSD schema

- An XSD schema can be generated from Java classes



JAXB: object to XML

```
try {
	Table table = new Table("GREEN", 10, 20);
	File file = new File("new_table.xml");
	JAXBContext jaxbContext =
		JAXBContext.newInstance(Table.class);
	Marshaller jaxbMarshaller =
		jaxbContext.createMarshaller();

	jaxbMarshaller.setProperty(
		Marshaller.JAXB_FORMATTED_OUTPUT, true);
	jaxbMarshaller.marshal(table, file);
	jaxbMarshaller.marshal(table, System.out);
} catch (JAXBException e) {
	e.printStackTrace();
}
```

JAXB: XML to object

```
try {
	File file = new File("table.xml");
	JAXBContext jaxbContext =
		JAXBContext.newInstance(Table.class);
	Unmarshaller jaxbUnmarshaller =
		jaxbContext.createUnmarshaller();
	Table table = (Table) jaxbUnmarshaller.unmarshal(
			file);
	System.out.println(table);

} catch (JAXBException e) {
	e.printStackTrace();
}
```

Date and Time

Date and Time API

- The Java Date and Time API provides classes for working with dates and times in Java
- Before JDK 8 the API was quite limited
- As of the JDK 8 a new date and time API based on the Joda-Time library is included in the JDK
- Root package of the new date and time API is **java.time**

Date and Time API

- Main classes of the Date and Time API (pre-JDK 8) include:

java.util.Date	Represents date and time
java.util.Calendar	Provides arithmetic operation on dates
java.util.GregorianCalendar	Represents Gregorian calendar
java.util.TimeZone	Represents a time zone and provides operations on time zones

- Example (getting the current date):

```
System.currentTimeMillis() // current date in milliseconds
```

```
Date date = new Date(); // current date as a Date instance
```

Date and Time API

- Main classes of the new Date and Time API (as of JDK 8) include:

java.time.Clock	Represents a date in a timezone
java.time.Instant	Represents instantaneous point in time
java.time.Duration	Represents duration
java.time.LocalDate	Represents a date without timezone (ISO-8601)
java.time.LocalTime	Represents a time without timezone (ISO-8601)
java.time.LocalDateTime	Represents a date-time without timezone
java.time.OffsetDateTime	Represents a date time with UTC/Greenwich offset
MonthDay Year YearMonth	Month day (ISO-8601) Year (ISO-8601) Year-Month (ISO-8601)
ZoneId	A time zone ID such as Europe/Sofia

Full list: https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html

New Date and Time API

```
Clock clock = Clock.systemUTC();
ZoneId zone = ZoneId.systemDefault();
ZoneId customZone = ZoneId.of("Europe/Berlin");
Clock customClock = Clock.system(customZone);
DateTimeFormatter formatter = DateTimeFormatter.
       ofPattern("MM-DD-YYYY");
LocalDate currentDate = LocalDate.now();
LocalDate date = LocalDate.of(2014, Month.JUNE, 10);
date = date.withYear(2015); // 2015-06-10
date = date.plusMonths(2); // 2015-08-10
date = date.minusDays(1);
String formattedDate = date.format(formatter);
System.out.println(formattedDate);
```

New Date and Time API

```
LocalTime time = LocalTime.of(20, 30);
time = time.withSecond(6); // 20:30:06
time = time.plusMinutes(3); // 20:33:06

LocalDateTime first = LocalDateTime.of(2014,
        Month.JUNE, 10, 20, 30);
LocalDateTime second = LocalDateTime.of(date,
       time);
// instant is the equivalent of java.util.Date
Instant instant = Instant.now();
instant.get(ChronoField.MONTH_OF_YEAR);
instant.plus(6, ChronoUnit.YEARS);
Instant second = Instant.now();
Duration duration = Duration.between(instant,
       second);
```

New Date and Time API

```
ZoneId zone = ZoneId.of("Europe/Paris");
LocalDate date = LocalDate.of(2014, Month.JUNE, 10);
ZonedDateTime zdt1 = date.atStartOfDay(zone);
Instant instant = Instant.now();
ZonedDateTime zdt2 = instant.atZone(zone);
```

New Date and Time API

- Reasons for the new date and time API in JDK 8:
  - Date is mutable
  - Date is not a “date”, but rather an instantaneous point in time
  - Dates are constructed with Calendar, but formatters work with Date
  - Limited set of operations for working with dates

The new date and time API should be preferred instead of the old one

Locales

Internationalization (i18n)

- Locales are basis for internationalization (i18n) capabilities provide by the Java platform
- Internationalization provides the ability to support multiple languages and regions within an application
- Internationalization extracts text constants from source code so that they can be translated into different languages

Hardcoding text that might be displayed to the user in i18n form (translated to a different language) needs to be refactored

No i18n support

- Consider the following basic example:

```
System.out.println("Some text in English");
```

- The above statements prints a text only in English …

- To make it internationalized we need to extract the hardcoded text in a properties file (one per language) and read the text in the corresponding language from there

i18n support

- The internationalized version of the previous example:

```
public static String getLocalizedText(String country,
       String language) {
       Locale locale =
              new Locale(language, country);
       ResourceBundle messages =
              ResourceBundle.getBundle("messages",
              locale);
       return messages.getString("sample_message");
}
```

```
System.out.println(getLocalizedText("BG", "bg"));
```

messages.properties

```
sample_message=Some text in English
```

messages_bg_BG.properties

```
sample_message=Някакъв текст на български
```

Locale

- The **java.util.Locale** class is the central unit of internalization support provided by the Java language
- From the Javadoc: *“a Locale object represents a specific geographical, political, or cultural region.”*
- A local can be created in any of the following ways:
  - with the Locale constructors
  - with the Locale.Builder class
  - with the Local.forLanguageTag factory method
  - with the constants in the Locale class

Locale

```
Locale bgLocale = new Locale.Builder()
       .setLanguage("bg")
       .setRegion("BG").build();
```

```
Locale bgLocale = new Locale("bg", "BG");
```

```
Locale bgLocale = Locale.forLanguageTag("bg-BG");
```

```
Locale bgLocale = Locale.forLanguageTag("bg-BG");c
```

```
Locale deLocale = Locale.GERMAN;
```

Locale API

- Useful methods of the Locale class include:

getAvailableLocales()	Returns an array of available (installed) locales
getISOCountries()	Returns an array of countries as per ISO
getISOLanguages()	Returns an array of languages as per ISO
set() setInt()/setLong() …	Sets a value for the field on a specified instance. Setter methods are available for fields of primitive type
getCountry() getDisplayCountry()	Returns the country code/localized name
getLanguage() getDisplayLanguage()	Returns the language code/localized name

i18n support

- Text is not the only element that may need internationalization support …
- Numbers and currencies may be formatted differently according the country and region
- Dates and times can also be formatted differently according to country and region

Formatting numbers

- Numbers can be formatted according to a locale using the **java.text.NumberFormat** class

```
int count = 345;
double price = 111.11;
Locale bgLocale =  new Locale("bg", "BG");
NumberFormat formatter =
        NumberFormat.getNumberInstance(bgLocale);
System.out.println(formatter.format(count)); // 345
System.out.println(formatter.format(price)); // 111,11
```

Formatting currencies

- Currencies can also be formatted according to a locale using the **java.text.NumberFormat** class

```
double price = 111.11;
Locale bgLocale =  new Locale("bg", "BG");
NumberFormat formatter =
        NumberFormat.getCurrencyInstance(bgLocale);
System.out.println(formatter.format(price));// 111,11 лв.
```

Formatting dates

- Dates can be formatted according to a locale using the **java.text.DateFormat** class

```
Locale bgLocale =  new Locale("bg", "BG");
DateFormat formatter =
      DateFormat.getDateInstance(DateFormat.DEFAULT,
      bgLocale);
Date today = new Date();
System.out.println(
      formatter.format(today)); // 19.03.2020 г.
```

Formatting times

- Times can also be formatted according to a locale using the **java.text.DateFormat** class

```
Locale bgLocale =  new Locale("bg", "BG");
DateFormat formatter =
       DateFormat.getTimeInstance( DateFormat.DEFAULT,
       bgLocale);
Date today = new Date();
System.out.println(
       formatter.format(today)); // 13:55:47 ч.
```

Using custom formats

- Numbers, currencies, dates and times can also be formatted according to a custom pattern and a specified locale

```
DecimalFormat formatter = new
        DecimalFormat("BG###,###.### lv");
System.out.println(
       formatter.format(456111.123)); // BG456 111,123 lv
```

```
SimpleDateFormat formatter =
       new SimpleDateFormat("yyyy-MM-dd");
System.out.println(formatter.format(new Date()));
```

DecimalFormat and SimpleDateFormat work also with locales

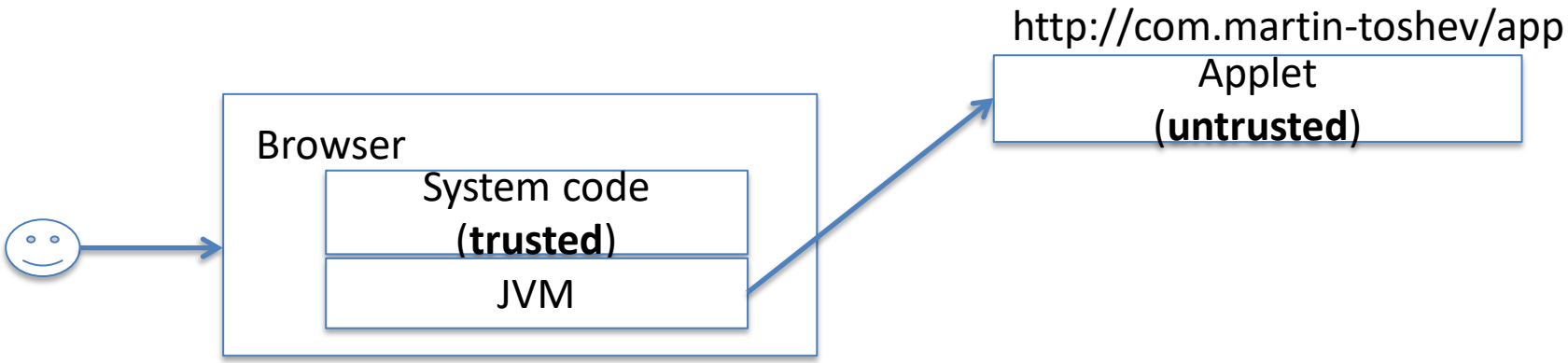
Security

The Java security model

- Traditionally - companies protect they assets using strict physical and network access policies
- Tools such as anti-virus software, firewalls, IPS/IDS systems facilitate this approach
- With the introduction of various technologies for loading and executing code on the client machine from the browser (such as Applets) a new range of concerns emerge related to client security

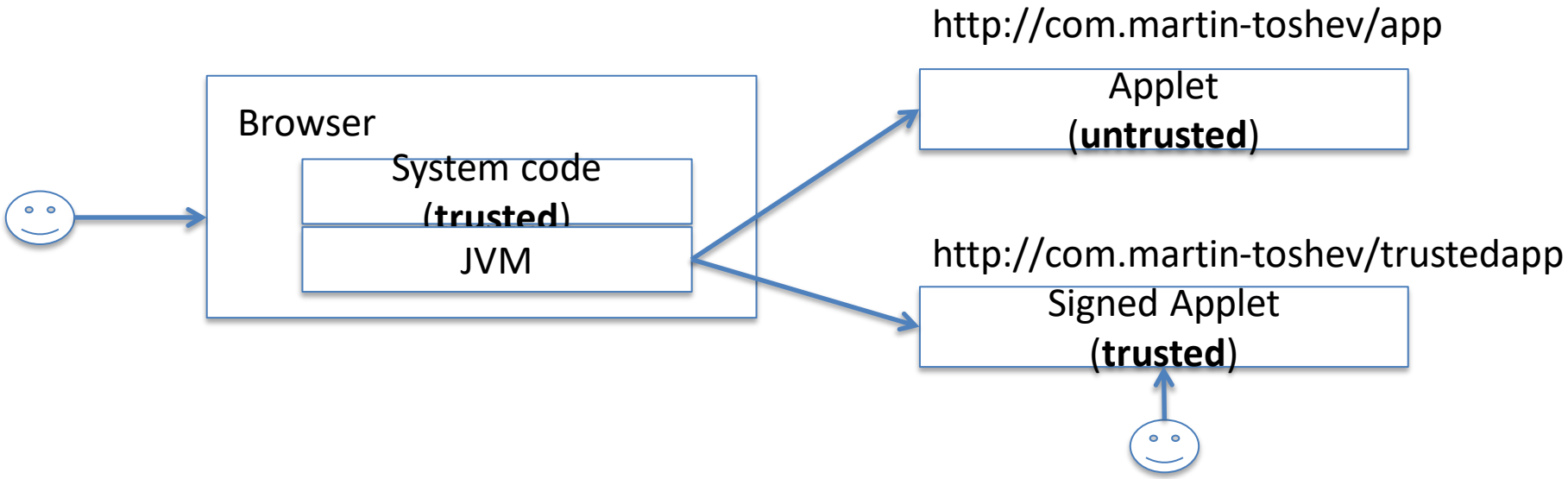
The Java security model

- **JDK 1.0** (when it all started …) – the original sandbox model was introduced



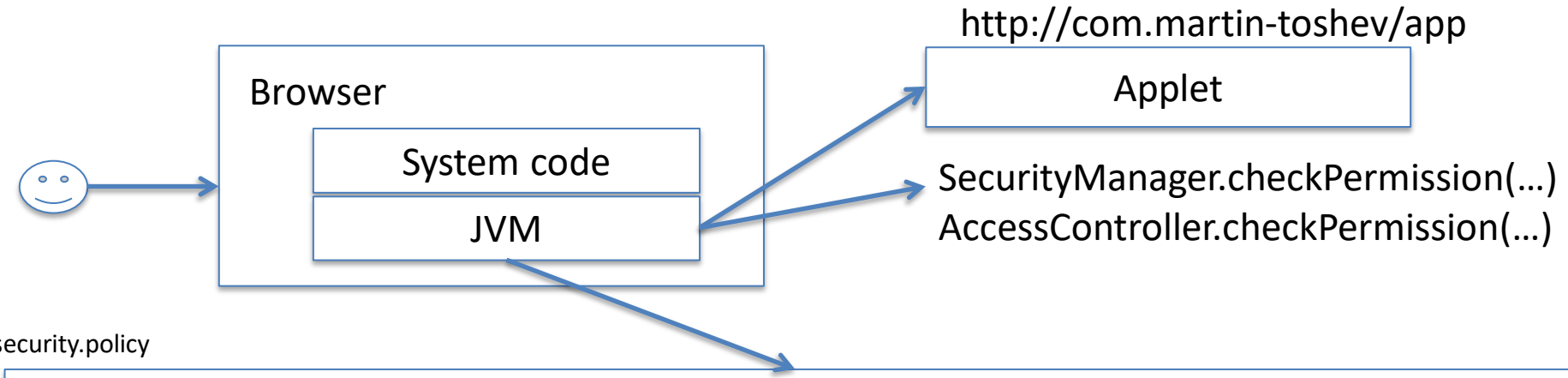
The Java security model

- **JDK 1.1** (gaining trust …) – applet signing introduced



The Java security model

- **JDK 1.2** (gaining more trust …) – fine-grained access control



security.policy

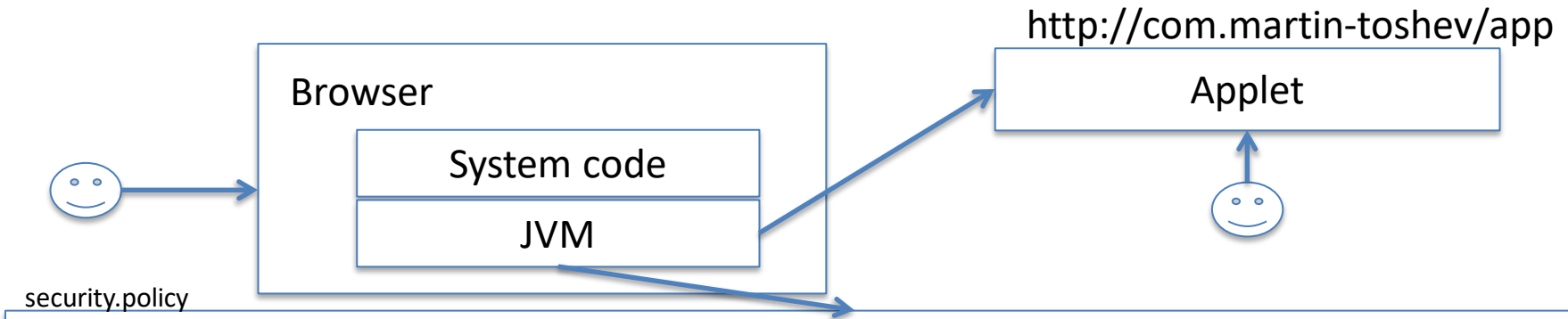
```
grant codeBase http://com.martin-toshev/app {
      permission java.io.FilePermisions "C:\\Windows" "delete"
}
```

The Java security model

- The notion of protection domain introduced – determined by the security policy
- Two types of protection domains – system and application
- The security model becomes code-centric
- No more notion of trusted and untrusted code

The Java security model

- **JDK 1.3, 1,4** (what about entities running the code … ?) – JAAS



The Java security model

- JAAS (Java Authentication and Authorization Service) extends the security model with role-based permissions
- The protection domain of a class now may contain not only the code source and the permissions but a list of principals
- The authorization component of JAAS extends the syntax of the Java security policy

The Java security model

- Up to JDK 1.4 the following is a typical flow for permission checking:
  - upon system startup a security policy is set and a security manager is installed

```
Policy.setPolicy(…)
System.setSecurityManager(…)
```

  - during classloading (e.g. of a remote applet) bytecode verification is done and the protection domain is set for the current classloader (along with the code source, the set of permissions and the set of JAAS principals)

The Java security model

- Up to JDK 1.4 the following is a typical flow for permission checking:
  - when system code is invoked from the remote code the SecurityManager is used to check against the intersection of protection domains based on the chain of threads and their call stacks

```
SocketPermission permission = new
       SocketPermission("martin-Toshev.com:8000-9000",
              "connect,accept");
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(permission);
```

  - application code can also do permission checking against remote code using a SecurityManager or an AccessController

The Java security model

- Up to JDK 1.4 the following is a typical flow for permission checking:

5) application code can also do permission checking with all permissions of the calling domain or a particular JAAS subject

```
AccessController.doPrivileged(…)
Subject.doAs(…)
Subject.doAsPrivileged(…)
```

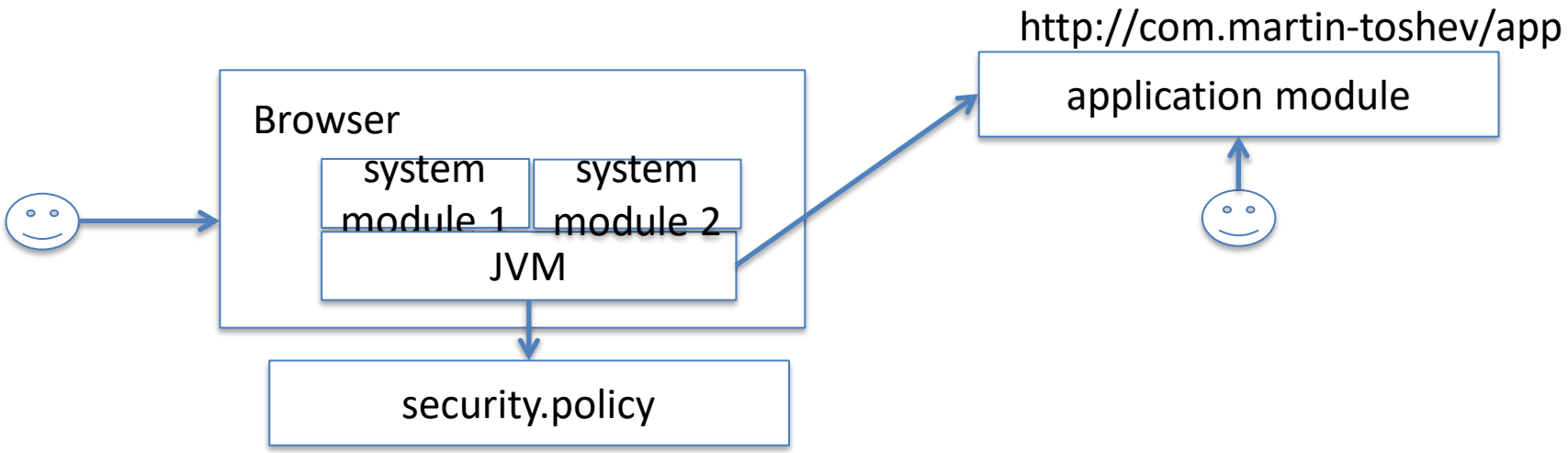
The Java security model

- **JDK 1.5, 1.6** (enhancing the model …) – new additions to the sandbox model (e.g. LDAP support for JAAS)

- **JDK 1.7, 1.8** (further enhancing the model …) – enhancements to the sandbox model (e.g. AccessController.doPrivileged() for checking against a subset of permissions)

The Java security model

- **JDK 1.9 and later … (applying the model to modules …)**



GUI programming

The Java security model

- The Java platform provides three main technologies for development of desktop applications:
  - AWT (Abstract Web Toolkit)
  - Swing (the successor of AWT)
  - JavaFX
- Of the three only JavaFX is not included as part of the JDK

We are not covering sessions on the Java GUI frameworks in academy

Swing

- Example:

```
SwingUtilities.invokeLater(new Runnable() {
        public void run() {
                MainFrame frame = new MainFrame();
                try {
                        frame.init();
                } catch (IOException e) {
                        LOGGER.log(Level.SEVERE,
                                e.getMessage(), e);
                }
        }
});
```

You can review the follow Java security tutorial written with Swing:
https://github.com/martinfmi/java_security_animated

Questions ?