

JC2002 Java Programming

Day 5: Testing and memory concepts (CS)

Monday, 6 November

JC2002 Java Programming

Day 5, Session 1: Testing and dependency management

Learning objectives

- After today's session, you should be able to:
 - Explain the purpose of testing in software development
 - Follow test-driven software development process in your project
 - Implement simple automated tests in Maven
 - Manage dependencies in your projects
 - Explain the basic memory concepts, such as difference between destructive and nondestructive process, as well as pass-by-value and pass-by-reference
 - Implement algorithms using recursion

Good coding practice

- Following good coding practice helps to avoid mistakes and makes it easier to find possible bugs
 - Indent your code properly, use parenthesis and curly brackets consistently
 - Name your classes and variables logically and consistently
 - Make sure variables are always properly initialised
 - Use comments and optional annotations (such as `@Override`) to clarify any potential confusion in your program code
 - **Test your code rigorously!**

Software testing

- Software testing is the process of verifying that the code is functioning correctly and producing the expected output
- Testing is an essential part of the software development life cycle
 - Testing ensures your program works as it is supposed to work
 - Testing helps to catch bugs and issues early in the development process, reducing the cost and time of fixing them later
 - Testing helps to keep software design modular so that the units can be tested in a meaningful way; if you are not able to test your class properly, your class design is maybe not appropriate

Levels of software testing

- Software can be tested at different levels, requiring different testing methodologies
 - **Unit testing:** testing units (classes or methods etc.) in isolation
 - **Integration testing:** testing two or more units together
 - **Functional testing:** testing specific functionality of an app
 - **System testing:** testing interfaces and functionality from end-user perspective
 - **System integration testing:** testing collaborating applications
- In this course, we focus on *unit testing*

Other types of software testing

- In this course, we focus on testing the code to make sure that it gives expected outputs and the program does not crash unexpectedly etc.; however, other types of tests also exist
 - **Performance testing:** testing on processor and network load, handling throughput, memory etc.
 - **Security testing:** testing application's robustness against security vulnerabilities and secure handling of data
 - **Usability testing:** testing that the applications is easy and logical to use for the end users
 - etc.

Writing unit tests

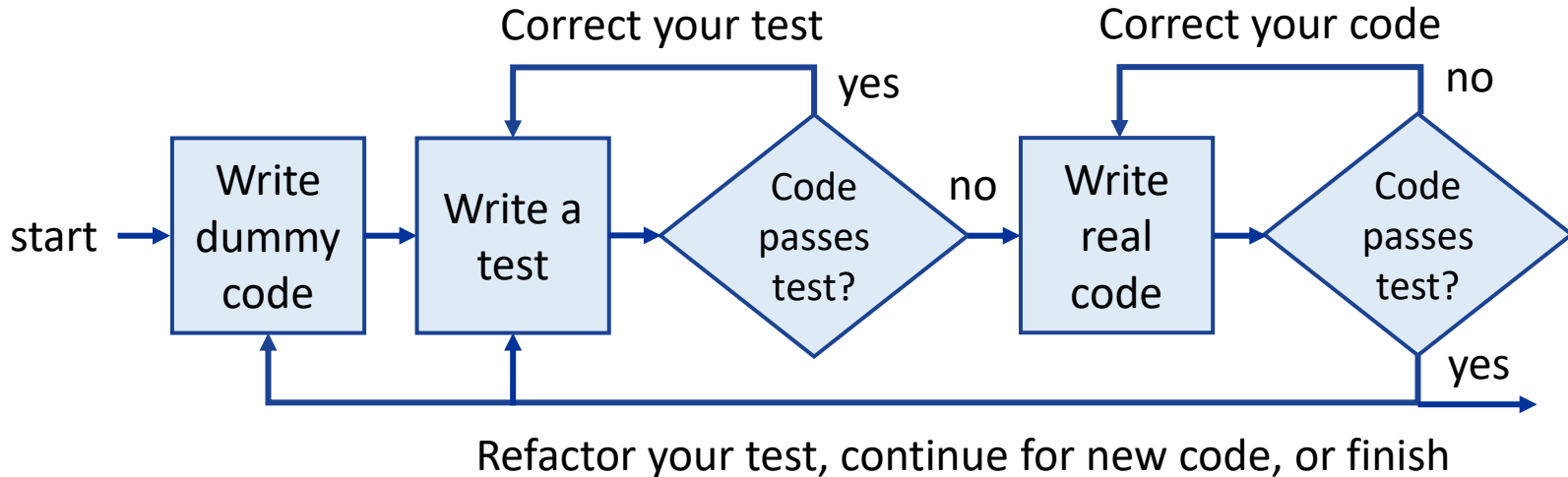
- Developers create test cases that cover all possible inputs and outputs of a particular unit of code
 - Test cases should include typical input, edge cases, unexpected behaviour, and handling of errors and exceptions
 - It can be time consuming and it may require significant amounts of effort from the developers!
- Unit tests are often automated using testing frameworks like Maven
 - Even though automated tests take more time for the initial setup, it saves time in a long run
 - In some cases, it is still the best option to run unit tests manually

Benefits of automated tests

- Automated testing helps reduce the time and effort required to test software applications, since the same tests can run multiple times without the need for human intervention
 - Automated tests run faster
 - Automated tests can be run more frequently
 - Automated tests free up testers to focus on more complex or exploratory testing tasks
 - Automated tests do not forget about what is left to test
 - Automated tests do not get bored!
- We will talk about automated testing with Maven in the next session

Test-driven development

- Using tests to drive the implementation of the code



Example of test-driven development (1)

- Let us assume that we want to write a method to compute number of combinations, i.e.,

$$C(n,k) = n! / r!(n-r)!$$

Combinations.java

```
1 // class for computing combinations
2 public class Combinations {
3     public int computeC(int n, int r) {
4         return -1; // indicates error
5     }
6 }
```

TestCombinations.java

```
1 public class TestCombinations {
2
3     // define test cases
4     static int[] n = {5,2,5,100};
5     static int[] r = {3,1,-1,50};
6
7     public static void main(String[] args) {
8
9         // run test cases
10        for(int i=0; i<n.length; i++) {
11            Combinations comb = new Combinations();
12            System.out.printf("n=%d, r=%d, C(n,r)=%d\n",
13                n[i],r[i],comb.computeC(n[i],r[i]));
14        }
15    }
16 }
```

Example of test-driven development (2)

Combinations.java

```
1 // Class for computing combinations
2 public class Combinations {
3     public int computeC(int n, int r) {
4         return -1; // indicates error
5     }
6 }
```

We first write a dummy method that always returns -1 (error)

Test fails, because it does not give the right answers

TestCombinations.java

```
1 public class TestCombinations {
2
3     // define test cases
4     static int[] n = {5,2,5,100};
5     static int[] r = {3,1,-1,50};
6
7     public static void main(String[] args) {
8
9         // run test cases
10        for(int i=0; i<n.length; i++) {
11            Combinations comb = new Combinations();
12            System.out.printf("n=%d, r=%d, C(n,r)=%d\n",
13                n[i],r[i],comb.computeC(n[i],r[i]));
14        }
15    }
16 }
```

```
n=5, r=3, C(n,r)=-1
n=2, r=2, C(n,r)=-1
n=5, r=-1, C(n,r)=-1
n=100, r=50, C(n,r)=-1
```

Example of test-driven development (3)

Combinations.java

```
1 // Class for computing combinations
2 public class Combinations {
3     private int fact(int x) {
4         int y = 1;
5         for(int i=1; i<=x; y *= i++);
6         return y;
7     }
8     public int computeC(int n, int r) {
9         int res = fact(n)/(fact(r)*fact(n-r));
10        return res; // return result
11    }
12 }
```

Write real functionality

TestCombinations.java

```
1 public class TestCombinations {
2
3     // define test cases
4     static int[] n = {5,2,5,100};
5     static int[] r = {3,1,-1,50};
6
7     public static void main(String[] args) {
8
9         // run test cases
10        for(int i=0; i<n.length; i++) {
11            Combinations comb = new Combinations();
12            System.out.printf("n=%d, r=%d, C(n,r)=%d%n",
13                              n[i],r[i],comb.computeC(n[i],r[i]));
14        }
15    }
16 }
```

Example of test-driven development (4)

Combinations.java

```
1 // Class for computing combinations
2 public class Combinations {
3     private int fact(int x) {
4         int y = 1;
5         for(int i=1; i<=x; y *= i++);
6         return y;
7     }
8     public int computeC(int n, int r) {
9         int res = fact(n)/(fact(r)*fact(n-r));
10        return res; // return result
11    }
12 }
```

Correct results

Illegal input, should return -1!

Division by zero. Why is that?

TestCombinations.java

```
1 public class TestCombinations {
2
3     // define test cases
4     static int[] n = {5,2,5,100};
5     static int[] r = {3,2,-1,50};
6
7     public static void main(String[] args) {
8
9         // run test cases
10        for(int i=0; i<n.length; i++) {
11            Combinations comb = new Combinations();
12            System.out.printf("n=%d, r=%d, C(n,r)=%d\n",
13                n[i],r[i],comb.computeC(n[i],r[i]));
14        }
15    }
16 }
```

n=5, r=3 C(n,r)=10

n=2, r=2 C(n,r)=1

n=5, r=-1 C(n,r)=0

Exception: / by zero

Example of test-driven development (5)

Combinations.java

```
1 public class Combinations {
2     private long fact(long x) {
3         long y = 1;
4         for(long i=1; i<=x; y *= i++);
5         return y;
6     }
7     public long computeC(long n, long r) {
8         if(n<1 || r<1 || r>n || n>20) {
9             return -1; // illegal input
10        }
11        long res = fact(n)/(fact(r)*fact(n-r));
12        return res; // return result
13    }
14 }
```

Test for out-of-bounds input

Use **long** to handle larger values

TestCombinations.java

```
1 public class TestCombinations {
2
3     // define test cases
4     static int[] n = {5,2,5,100};
5     static int[] r = {3,1,-1,50};
6
7     public static void main(String[] args) {
8
9         // run test cases
10        for(int i=0; i<n.length; i++) {
11            Combinations comb = new Combinations();
12            System.out.printf("n=%d, r=%d, C(n,r)=%d\n",
13                n[i],r[i],comb.computeC(n[i],r[i]));
14        }
15    }
16 }
```

Example of test-driven development (6)

Combinations.java

```
1 public class Combinations {
2     private long fact(long x) {
3         long y = 1;
4         for(long i=1; i<=x; y *= i++);
5         return y;
6     }
7     public long computeC(long n, long r) {
8         if(n<1 || r<1 || r>n || n>20) {
9             return -1; // illegal input
10        }
11        long res = fact(n)/(fact(r)*fact(n-r));
12        return res; // return result
13    }
```

Now it works but note that max. value of n is 20 (even long cannot compute larger factorials); better implementations could be made!

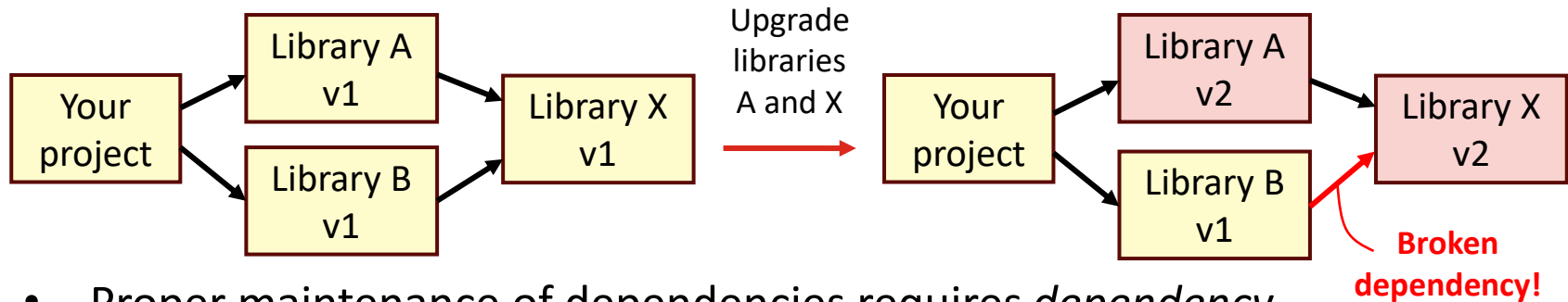
TestCombinations.java

```
1 public class TestCombinations {
2
3     // define test cases
4     static int[] n = {5,2,5,100};
5     static int[] r = {3,1,-1,50};
6
7     public static void main(String[] args) {
8
9         // run test cases
10        for(int i=0; i<n.length; i++) {
11            Combinations comb = new Combinations();
12            System.out.printf("n=%d, r=%d, C(n,r)=%d\n",
13                n[i],r[i],comb.computeC(n[i],r[i]));
14        }
15    }
16 }
```

```
n=5, r=3, C(n,r)=10
n=2, r=2, C(n,r)=1
n=5, r=-1, C(n,r)=-1
n=100, r=50, C(n,r)=-1
```


Dependency management

- Often large software project use different third-party libraries
 - There are different (sometimes complex) *dependencies* between libraries: certain library may use a specific version of another library
 - When upgrading to new versions, dependencies can get broken



- Proper maintenance of dependencies requires *dependency management*

Basics of dependency management

- When you upgrade a third-party library, you possibly need to make some changes in your code using that library
 - See the API documentation of the library
- In case of dependency-related errors, you need to find the library that produces the error
 - The error message often shows where the problem occurs, e.g.:
`java.lang.NoClassDefFoundError: ch/qos/logback/core/status/WarnStatus`
 - Online tool serfish (<https://serfish.com/jar/>) can be useful for finding `.jar` packages for missing classes
 - It is recommended to use dependency management tools, e.g., Maven

Questions, comments?

JC2002 Java Programming

Day 5, Session 2: Introduction to Maven

What is Apache Maven?

- Project management software making developer's life *much* easier with the following objectives:
 - Making the build process easy
 - Providing a uniform build system
 - Providing quality project information
 - Encouraging better development practices
- Website for more information: <https://maven.apache.org/>
- Other projects similar to Maven:
 - Gradle (<https://gradle.org/>)
 - Ant (<https://ant.apache.org/>)

What will Maven help you with?

- Builds
- Documentation
- Reporting
- Dependencies
- Source code management (SCM)
- Releases
- Distribution

Set up Maven

- In your Codio box, open terminal and check if Maven is installed:

```
mvn -version
```

- If it is not installed (i.e., you got an error “mvn not found”), install it in your terminal:

```
apt update
```

```
apt install maven
```

Building a default project in Maven

- Maven *archetypes* are template projects that you can base your own projects on
- Type `mvn archetype:generate` to generate a “HelloWorld” project based on a "quickstart" archetype
 - Enter 2082 (or whatever is the default)
 - Pick archetype version (options given, you can use the default)
 - Enter groupId (usually identifier for the organisation)
 - Enter artifactId (*jar* file name without version, e.g., *myapp*)
 - Enter the snapshot version (e.g. 1.0)
 - Enter the package name (by default the same as groupId)

Dynamic binding

- Maven's configuration file defines the *Project Object Model* (POM)
- Key elements (source: <https://maven.apache.org/guides/getting-started/index.html>):
 - **project**: This is the top-level element in all Maven *pom.xml* files.
 - **modelVersion**: This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.
 - **groupId**: This element indicates the unique identifier of the organization or group that created the project. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example, org.apache.maven.plugins is the designated groupId for all Maven plugins.



Elements in pom.xml (continues)

- **artifactId:** This element indicates the unique base name of the primary artifact being generated by this project. The primary artifact for a project is typically a JAR file. Secondary artifacts like source bundles also use the artifactId as part of their final name. A typical artifact produced by Maven would have the form `<artifactId>-<version>.<extension>` (for example, `myapp-1.0.jar`).
- **version:** This element indicates the version of the artifact generated by the project.
- **name:** This element indicates the display name used for the project. This is often used in Maven's generated documentation.
- **url:** This element indicates where the project's site can be found. This is often used in Maven's generated documentation.
- **properties:** This element contains value placeholders accessible anywhere within a POM.
- **dependencies:** This element's children list dependencies. The cornerstone of the POM.
- **build:** This element handles things like declaring your project's directory structure and managing plugins.

Building a default project in Maven

- No need to compile files individually, just execute:

```
mvn compile
```

- Generate `.jar` file by executing:

```
mvn package
```

and find the JAR file in the *target* directory. Then, run JAR file:

```
java -jar <file name>.jar
```

- **Note:** To make JAR files executable, you must edit the manifest with the information about the class that contains the main method which you want to run (see the next slide)

Adding manifest information in pom.xml

- The manifest is a special block that can contain information about the files packaged in a JAR file (e.g., class to run first)

```
<plugin>
<artifactId>maven-jar-plugin</artifactId>
<version>3.0.2</version>
<configuration>
  <archive>
    <manifest>
      <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
      <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>
      <addClasspath>true</addClasspath>
      <mainClass>org.test.App</mainClass>
    </manifest>
  </archive>
</configuration>
</plugin>
```

Look for this placeholder in the pom.xml file

The main class of your application

Add this part in the file

Dependency management using pom.xml

- Add references to dependencies in pom.xml
 - Add as many dependencies as you need using `<dependency>` ... `</dependency>` tags
- To see the dependency tree, use:
mvn dependency:tree
- Repository of Java open source libraries managed with Maven:
<https://mvnrepository.com/>

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    ...
  </dependency>
</dependencies>
```

Testing your projects with JUnit and Maven

- Maven helps you to automate testing of your code
 - Maven promotes best practices by keeping tests separate from the source code
 - Maven streamlines the execution of tests and reporting of test results
- JUnit is a testing framework for JVM, providing API for writing test cases in Java code
 - Maven is then used to execute the JUnit tests

Default JUnit test

- The *src/test/java* directory contains the source code of our unit tests
- If we use Maven quickstart archetype, an example test will be created automatically (file *AppTest.java*)

AppTest.java

```
1 package myorg;
2
3 import static org.junit.Assert.assertTrue;
4
5 import org.junit.Test;
6
7 /**
8  * Unit test for simple App.
9  */
10 public class AppTest
11 {
```

```
12     /**
13      * Rigorous Test :-)
14      */
15     @Test
16     public void shouldAnswerwithTrue()
17     {
18         assertTrue( true );
19     }
20 }
```

Writing JUnit tests

- Automated testing helps you ensure that the code is running as expected even after you make changes later
- *Unit tests* focus on the individual code components (units), not on the external environment, end to end system testing, etc.
- Use assert statements to check if methods return expected outcomes (<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>)
- Testing can get complex and multiple tests may be needed to test a single method
- *Lack of test writing skills is one of the most frequent complaints from organisations in relation to new graduates entering the job market!*

Example test: App.java

- We want to test two methods: **calculateSum()** and **printSum()**

App.java

```
1  package myorg;
2
3  // Hello world!
4  public class App
5  {
6      public int calculateSum(int a, int b) {
7          return a + b;
8      }
9      public void printSum(int a) {
10         System.out.println("The sum is: " + a);
11     }
12     public static void main( String[] args )
13     {
14         App app = new App();
15         app.printSum(app.calculateSum(1,2));
16     }
17 }
```

Example test: AppTest.java

AppTest.java

```
1  package myorg;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5
6  import java.io.ByteArrayOutputStream;
7  import java.io.PrintStream;
8
9  // Unit test for simple App.
10 public class AppTest
11 {
12     private final ByteArrayOutputStream
13         outContent = new ByteArrayOutputStream();
14     private final PrintStream
15         originalOut = System.out;
16
17     // Test printing
18     @Test
19     public void testPrinting()
20     {
```

```
21         System.setOut(new PrintStream(outContent));
22         App obj = new App ();
23         obj.printSum(3);
24         assertEquals("The sum is: 3",
25             outContent.toString().trim());
26         System.setOut(originalOut);
27     }
28
29     // Test addition
30     @Test
31     public void addNumbers()
32     {
33         App obj = new App ();
34         assertEquals( 2, obj.calculateSum(1,1));
35         assertEquals( -2, obj.calculateSum(-1,-1));
36         assertEquals( 0, obj.calculateSum(0,0));
37     }
38 }
```

Example test: AppTest.java

AppTest.java

```
1  package myorg;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5
6  import java.io.ByteArrayOutputStream;
7  import java.io.PrintStream;
8
9  // Unit test for simple App.
10 public class AppTest
11 {
12     private final ByteArrayOutputStream
13         outContent = new ByteArrayOutputStream();
14     private final PrintStream
15         originalOut = System.out;
16
17     // Test printing
18     @Test
19     public void testPrinting()
20     {
```

```
21         System.setOut(new PrintStream(outContent));
22         App obj = new App ();
23         obj.printSum(3);
24         assertEquals("The sum is: 3",
25             outContent.toString().trim());
26         System.setOut(originalOut);
27     }
28
29     // Test addition
30     @Test
31     public void addNumbers()
32     {
33         App obj = new App ();
34         assertEquals( 2, obj.calculateSum(1,1));
35         assertEquals( -2, obj.calculateSum(-1,-1));
36         assertEquals( 0, obj.calculateSum(0,0));
37     }
38 }
```

Test for the edge cases, ensure your test has good coverage of the domain

Maven output of passed test

- Run `mvn clean test`

```
[INFO] -----  
[INFO]  T E S T S  
[INFO] -----  
[INFO] Running myorg.AppTest  
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.023 s - in myorg.AppTest  
[INFO]  
[INFO] Results:  
[INFO]  
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0  
[INFO]  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time:  2.182 s  
[INFO] Finished at: 2022-09-13T09:58:19Z  
[INFO] -----
```

Failed test example

- What happens if you change the test case in **AppTest.java** to produce a failed test case?

```
17  // Test printing
18  @Test
19  public void testPrinting()
20  {
21      System.setOut(new PrintStream(outContent));
22      App obj = new App ();
23      obj.printSum(3);
24      assertEquals("The sum is: 4",
25                  outContent.toString().trim());
26      System.setOut(originalOut);
27  }
```

Maven output of failed test

- Run `mvn clean test` again

```
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running myorg.AppTest
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.031 s <<< FAILURE! - in myorg.AppTest
[ERROR] testPrinting(myorg.AppTest) Time elapsed: 0.006 s <<< FAILURE!
org.junit.ComparisonFailure: expected:<The sum is: [4]> but was:<The sum is: [3]>
    at myorg.AppTest.testPrinting(AppTest.java:23)

[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   AppTest.testPrinting:23 expected:<The sum is: [4]> but was:<The sum is: [3]>
[INFO]
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 2.159 s
[INFO] Finished at: 2022-09-13T10:03:52Z
[INFO] -----
```

Questions, comments?

JC2002 Java Programming

Day 5, Session 3: Basic memory concepts


References and learning objects

- Today's remaining sessions are largely based on ***Java: How to Program***, Chapter 7, and ***Java in a Nutshell***
- After today's remaining sessions, you should be able to:
 - Explain the basic memory concepts, such as difference between destructive and nondestructive process, as well as pass-by-value and pass-by-reference
 - Implement algorithms using recursion

Memory concepts

- Every variable in Java has a *name*, a *type*, a *size* (in bytes), and a *value*
 - Name differentiates variables from each other
 - Size depends on type
 - Value can change during the execution of the program (unless the variable is a final variable)
- When a variable is initialised, JVM allocates space for the variable in the computer's working memory
 - When new value is assigned to the variable, the old value is lost (destructive process)
 - When the value of a variable is read, it can be used but it does not change (non-destructive process)

Memory concepts example



```
1 public class MemoryExample {  
2     public static void main(String[] args){  
3         int x = 5;  
4         int y = 10;  
5         int z = x + y;  
6         System.out.println("Result: " + z);  
7     }  
8 }
```

```
$ java MemoryExample
```

Program starts in the
beginning of method **main**

Memory

Memory concepts: initialise x

```
1 public class MemoryExample {  
2     public static void main(String[] args){  
3         int x = 5;  
4         int y = 10;  
5         int z = x + y;  
6         System.out.println("Result: " + z);  
7     }  
8 }
```

```
$ java MemoryExample
```

Space is allocated and value
is assigned to variable **x**

Memory

int x: 5

Memory concepts: initialise y

```
1 public class MemoryExample {  
2     public static void main(String[] args){  
3         int x = 5;  
4         int y = 10;  
5         int z = x + y;  
6         System.out.println("Result: " + z);  
7     }  
8 }
```

```
$ java MemoryExample
```

Space is allocated and value is assigned to variable **y**

Memory

int x: 5

int y: 10

Memory concepts: initialise z

```
1 public class MemoryExample {  
2     public static void main(String[] args){  
3         int x = 5;  
4         int y = 10;  
5         int z = x + y;  
6         System.out.println("Result: " + z);  
7     }  
8 }
```

Space is allocated and value is assigned to variable **z**, using variables **x** and **y**. This is a *nondestructive* memory process, since the values of **x** and **y** are not lost.

Memory

int z: 15

int x: 5

int y: 10

Memory concepts: use variable z

```
1 public class MemoryExample {  
2     public static void main(String[] args){  
3         int x = 5;  
4         int y = 10;  
5         int z = x + y;  
6         System.out.println("Result: " + z);  
7     }  
8 }
```

```
$ java MemoryExample  
Result: 15
```

Finally, value of **z**, is used as console output.
This is also a *nondestructive* memory process, since the value of **z** is not lost.

Memory

int z: 15

int x: 5

int y: 10

Memory concepts: execution ends


```
1 public class MemoryExample {  
2     public static void main(String[] args){  
3         int x = 5;  
4         int y = 10;  
5         int z = x + y;  
6         System.out.println("Result: " + z);  
7     }  
8 }
```

```
$ java MemoryExample  
Result: 15  
$
```

Execution of the program ends, and the working memory is cleared.

Memory

Memory concept example 2



```
1 public class MemoryExample2 {  
2     public static void main(String[] args){  
3         int x = 5;  
4         x = x + 10;  
5         System.out.println("Result: " + x);  
6     }  
7 }
```

```
$ java MemoryExample2
```

Also in the second example, program starts in the beginning of method **main**

Memory

Memory concepts 2: initialise x

```
1 public class MemoryExample2 {  
2     public static void main(String[] args){  
3         int x = 5;  
4         x = x + 10;  
5         System.out.println("Result: " + x);  
6     }  
7 }
```

```
$ java MemoryExample2
```

Space is allocated and value
is assigned to variable **x**

Memory

int x: 5

Memory concepts 2: reassign value to x

```
1 public class MemoryExample2 {  
2     public static void main(String[] args){  
3         int x = 5;  
4         x = x + 10;  
5         System.out.println("Result: " + x);  
6     }  
7 }
```

```
$ java MemoryExample2
```

New value is assigned to **x** and the old value is lost; this is a *destructive* memory process

Memory

int x: 15

Memory concepts 2: use value of x

```
1 public class MemoryExample2 {  
2     public static void main(String[] args){  
3         int x = 5;  
4         x = x + 10;  
5         System.out.println("Result: " + x);  
6     }  
7 }
```

```
$ java MemoryExample2  
Result: 15
```

Value of **x** is used but not changed

Memory

int x: 15

Memory concepts 2: execution ends

```
1 public class MemoryExample2 {  
2     public static void main(String[] args){  
3         int x = 5;  
4         x = x + 10;  
5         System.out.println("Result: " + x);  
6     }  
7 }
```

```
$ java MemoryExample2  
Result: 15
```

Execution ends and memory is cleared

Memory

Reference type variables

- Variables that are not primitive type variables are *reference type variables*: the variable contains a reference (pointer) to the memory location with the actual data
 - Arrays and references to objects are reference type variables
- Be careful to note the difference between modifying the variable (pointer) and modifying the data it refers to!

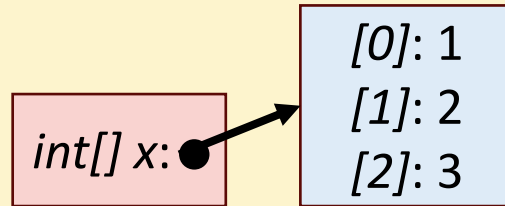
Reference type variable example (x[])

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample
```

Space is allocated for the reference variable **x** as well as the area containing the values of **x[0]**, **x[1]**, and **x[2]**

Memory



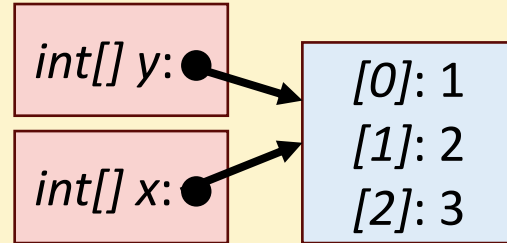
Reference type variable example (y[])

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample
```

Space is allocated for the reference variable **y**, but it is assigned the same value as **x**, so **x** and **y** refer to the same data!

Memory



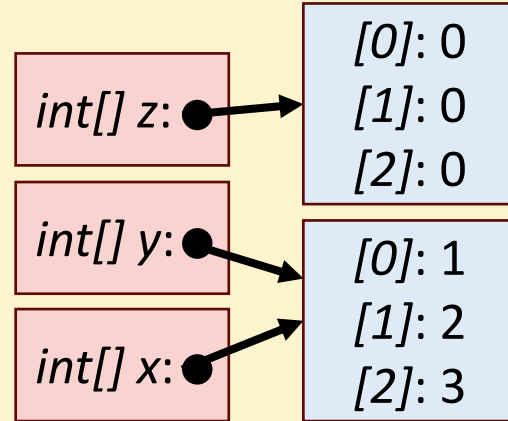
Reference type variable example (z[])

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample
```

Space is allocated for the reference variable **z** and the values of **z[0]**, **z[1]**, and **z[2]**

Memory



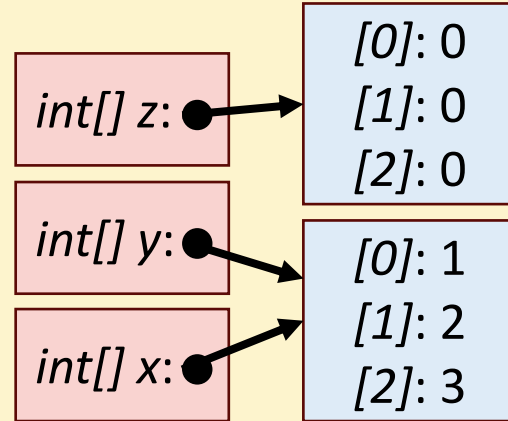
Reference type variable example (hashes)

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample  
Array x: 1510467688
```

We can print the hash codes, i.e., unique representations for memory locations, for each reference variable

Memory

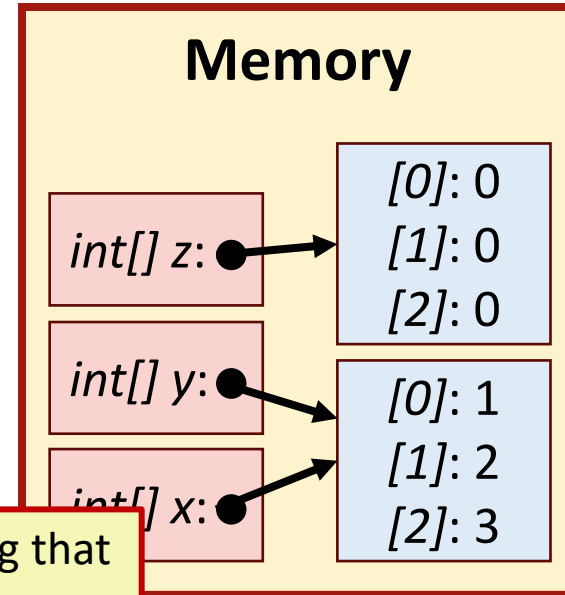


Reference type variable example (hashes)

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample  
Array x: 1510467688  
Array y: 1510467688
```

Hash codes for **x** and **y** are the same, indicating that they indeed refer to the same memory location

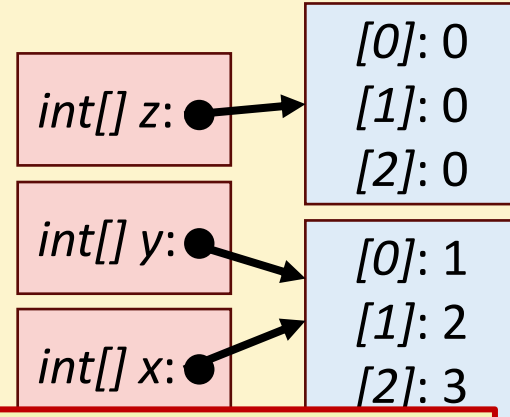


Reference type variable example (hashes)

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample  
Array x: 1510467688  
Array y: 1510467688  
Array z: 868693306
```

Memory



Hash code for **z** is different, indicating different memory location

Reference type variable example (end)

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x;  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample  
Array x: 1510467688  
Array y: 1510467688  
Array z: 868693306  
$
```

Execution ends and memory is cleared.

Memory

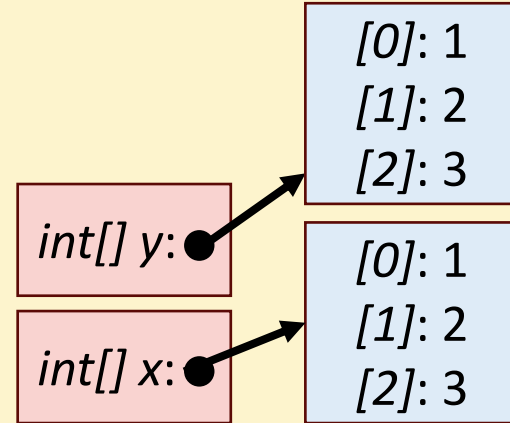
Deep copy method clone()

```
1 public class RefVarExample {  
2     public static void main(String[] args){  
3         int[] x = {1,2,3};  
4         int[] y = x.clone();  
5         int[] z = {0,0,0};  
6         System.out.println("Array x: " + x.hashCode());  
7         System.out.println("Array y: " + y.hashCode());  
8         System.out.println("Array z: " + z.hashCode());  
9     }  
10 }
```

```
$ java RefVarExample
```

If you want to make a full copy of the referred data and not just another reference, you can use method **clone()**

Memory



Pass-by-value vs. pass-by-reference

- In many programming languages (e.g., C++), there are two ways to pass arguments in method calls: *pass-by-value* (or *call-by-value*) and *pass-by-reference* (*call-by-reference*)
 - **Pass by value:** a copy of the argument's value is passed to the called method. The called method works exclusively with the copy. Changes to the copy do not affect the original variable's value.
 - **Pass by reference:** the called method can access the argument's value in the caller directly and modify that data, if necessary. This can improve performance by eliminating the need to copy large amounts of data.
- In Java, you cannot choose: *all arguments are passed by value*

Pass-by-value example (1)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Start program, assign values to array

Memory

main()

int[] array:

[0]: 1

[1]: 2

[2]: 3

Pass-by-value example (2)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Method main() calls method modifyArray()

Memory

modifyArray()

int[] arr:

[0]: 1

main()

int[] array:

[1]: 2

[2]: 3

Pass-by-value example (3)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Start a loop to modify elements of arr []

Memory

modifyArray()

int[] arr:

int i: 0

main()

int[] array:

[0]: 1

[1]: 2

[2]: 3

Pass-by-value example (4)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Modify arr[0]

Memory

modifyArray()

int[] arr:

int i: 0

main()

int[] array:

[0]: 2

[1]: 2

[2]: 3

Pass-by-value example (5)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Modify arr[1]

Memory

modifyArray()

int[] arr:

int i: 1

main()

int[] array:

[0]: 2

[1]: 4

[2]: 3

Pass-by-value example (6)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Modify arr[2]

Memory

modifyArray()

int[] arr:

int i: 2

main()

int[] array:

[0]: 2

[1]: 4

[2]: 6

Pass-by-value example (7)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Return to main(), array has been modified.
Call modifyElement() next.

Memory

main()

int[] array:

[0]: 2

[1]: 4

[2]: 6

Pass-by-value example (8)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Note that the passed variable *e* contains a *copy* of element `array[3]`!

Memory

`modifyElement()`
`int e: 4`

`main()`
`int[] array:`

`[0]: 2`
`[1]: 4`
`[2]: 6`

Pass-by-value example (9)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Value of e is modified, but the change is not visible in main()

Memory

modifyElement()

int e: 8

main()

int[] array: ●

[0]: 2

[1]: 4

[2]: 6

Pass-by-value example (10)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Return to main()

Memory

main()

int[] array:

[0]: 2

[1]: 4

[2]: 6

Pass-by-value example (11)

```
1 public class RefVarExample2 {  
2     public static void modifyArray(int[] arr) {  
3         for (int i = 0; i < arr.length; i++) {  
4             arr[i] *= 2;  
5         }  
6     }  
7     public static void modifyElement(int e) {  
8         e *= 2;  
9     }  
10    public static void main(String[] args) {  
11        int[] array = {1, 2, 3, 4, 5};  
12        modifyArray(array);  
13        modifyElement(array[3]);  
14    }  
15 }
```

Memory

Execution ends, memory is cleared

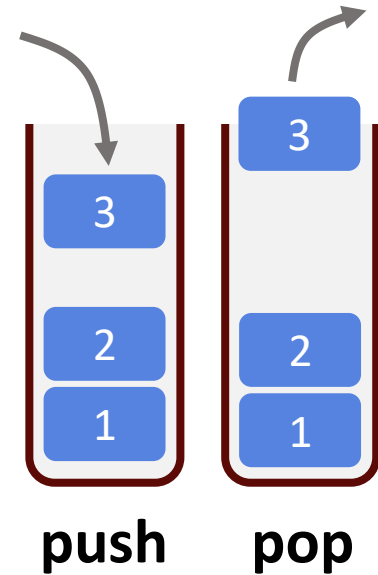
Questions, comments?

JC2002 Java Programming

Day 5, Session 4: Call stack and recursion

Call stack

- A *stack* is a data structure where data elements are piled on top of each other (like e.g., pile of cards)
 - The basic operations are *push*, which adds an element on the top, and *pop*, which removes the most recently added element from the top
- In JVM, call stack is used to keep track of method calls in memory
 - When a method is called, the local variables and other data of the method are pushed in the call stack
 - When the program returns from a method, the information is removed from the stack



Understanding call stack (1)

```
1 public class CallStackExample {  
2     private static int methodB(int x){  
3         int y = x-2;  
4         return y;  
5     }  
6     private static int methodA(int x){  
7         x = methodB(x);  
8         x *= 2;  
9         return x;  
10    }  
11    public static void main(String[] args){  
12        int x = 5;  
13        int y = methodA(x);  
14        System.out.println("Result: " + y);  
15    }  
16 }
```

Call stack

main()

Understanding call stack (2)

```
1 public class CallStackExample {
2     private static int methodB(int x){
3         int y = x-2;
4         return y;
5     }
6     private static int methodA(int x){
7         x = methodB(x);
8         x *= 2;
9         return x;
10    }
11    public static void main(String[] args){
12        int x = 5;
13        int y = methodA(x);
14        System.out.println("Result: " + y);
15    }
16 }
```

Call stack

main()
int x: 5

Understanding call stack (3)

```
1 public class CallStackExample {  
2     private static int methodB(int x){  
3         int y = x-2;  
4         return y;  
5     }  
6     private static int methodA(int x){  
7         x = methodB(x);  
8         x *= 2;  
9         return x;  
10    }  
11    public static void main(String[] args){  
12        int x = 5;  
13        int y = methodA(x);  
14        System.out.println("Result: " + y);  
15    }  
16 }
```

Call stack

methodA()
int x: 5

main()
int x: 5

Understanding call stack (4)

```
1 public class CallStackExample {  
2     private static int methodB(int x){  
3         int y = x-2;  
4         return y;  
5     }  
6     private static int methodA(int x){  
7         x = methodB(x);  
8         x *= 2;  
9         return x;  
10    }  
11    public static void main(String[] args){  
12        int x = 5;  
13        int y = methodA(x);  
14        System.out.println("Result: " + y);  
15    }  
16 }
```

Call stack

methodB()

int x: 5

methodA()

int x: 5

main()

int x: 5

Understanding call stack (5)

```
1 public class CallStackExample {  
2     private static int methodB(int x){  
3         int y = x-2;  
4         return y;  
5     }  
6     private static int methodA(int x){  
7         x = methodB(x);  
8         x *= 2;  
9         return x;  
10    }  
11    public static void main(String[] args){  
12        int x = 5;  
13        int y = methodA(x);  
14        System.out.println("Result: " + y);  
15    }  
16 }
```

Call stack

methodB()

int x: 5; int y: 3

methodA()

int x: 5

main()

int x: 5

Understanding call stack (6)

```
1 public class CallStackExample {  
2     private static int methodB(int x){  
3         int y = x-2;  
4         return y;  
5     }  
6     private static int methodA(int x){  
7         x = methodB(x);  
8         x *= 2;  
9         return x;  
10    }  
11    public static void main(String[] args){  
12        int x = 5;  
13        int y = methodA(x);  
14        System.out.println("Result: " + y);  
15    }  
16 }
```

Call stack

methodA()


int x: 6

main()

int x: 5

Understanding call stack (7)

```
1 public class CallStackExample {  
2     private static int methodB(int x){  
3         int y = x-2;  
4         return y;  
5     }  
6     private static int methodA(int x){  
7         x = methodB(x);  
8         x *= 2;  
9         return x;  
10    }  
11    public static void main(String[] args){  
12        int x = 5;  
13        int y = methodA(x);  
14        System.out.println("Result: " + y);  
15    }  
16 }
```



Call stack

main()

int x: 5; int y: 6

Using shadowed class attributes

- Method parameters are visible only within the method scope
- If a class attribute and a method parameter share the same name, Java will use the method parameter
- If you need to access the class attribute, use keyword **this**

Using keyword this: instantiate class

```
1 public class TestClass {  
2     private int x = 5;  
3     public TestClass() {}  
4     private int multiply(int y) {  
5         y = y * x;  
6         return y;  
7     }  
8     private void add(int x) {  
9         this.x += x;  
10        return;  
11    }  
12    public static void main(String[] args) {  
13        TestClass tc = new TestClass();  
14        int x = tc.multiply(2);  
15        tc.add(x);  
16        System.out.println("Result: " + tc.x);  
17    }  
18 }
```

Memory

TestClass object

int x: 5

main()

TestClass tc:



Using keyword this: instantiate class

```
1 public class TestClass {  
2     private int x = 5;  
3     public TestClass() {}  
4     private int multiply(int y) {  
5         y = y * x;  
6         return y;  
7     }  
8     private void add(int x) {  
9         this.x += x;  
10        return;  
11    }  
12    public static void main(String[] args) {  
13        TestClass tc = new TestClass();  
14        int x = tc.multiply(2);  
15        tc.add(x);  
16        System.out.println("Result: " + tc.x);  
17    }  
18 }
```

Memory

multiply()

int y: 2

TestClass this: ●

TestClass object

int x: 5

main()

TestClass tc: ●

Using keyword this: instantiate class

```
1 public class TestClass {  
2     private int x = 5;  
3     public TestClass() {}  
4     private int multiply(int y) {  
5         y = y * x;  
6         return y;  
7     }  
8     private void add(int x) {  
9         this.x += x;  
10        return;  
11    }  
12    public static void main(String[] args) {  
13        TestClass tc = new TestClass();  
14        int x = tc.multiply(2);  
15        tc.add(x);  
16        System.out.println("Result: " + tc.x);  
17    }  
18 }
```

Memory

multiply()

int y: 10

TestClass this: ●

TestClass object

int x: 5

main()

TestClass tc: ●

Using keyword this: instantiate class

```
1 public class TestClass {
2     private int x = 5;
3     public TestClass() {}
4     private int multiply(int y) {
5         y = y * x;
6         return y;
7     }
8     private void add(int x) {
9         this.x += x;
10        return;
11    }
12    public static void main(String[] args) {
13        TestClass tc = new TestClass();
14        int x = tc.multiply(2);
15        tc.add(x);
16        System.out.println("Result: " + tc.x);
17    }
18 }
```

Memory

TestClass object
int x: 5

main()
TestClass tc:
int x: 10

Using keyword this: instantiate class

```
1 public class TestClass {  
2     private int x = 5;  
3     public TestClass() {}  
4     private int multiply(int y) {  
5         y = y * x;  
6         return y;  
7     }  
8     private void add(int x) {  
9         this.x += x;  
10        return;  
11    }  
12    public static void main(String[] args) {  
13        TestClass tc = new TestClass();  
14        int x = tc.multiply(2);  
15        tc.add(x);  
16        System.out.println("Result: " + tc.x);  
17    }  
18 }
```

Memory

add()

int x: 10

TestClass this: ●

TestClass object

int x: 15

main()

TestClass tc: ●

int x: 10

Using keyword this: instantiate class

```
1 public class TestClass {  
2     private int x = 5;  
3     public TestClass() {}  
4     private int multiply(int y) {  
5         y = y * x;  
6         return y;  
7     }  
8     private void add(int x) {  
9         this.x += x;  
10        return;  
11    }  
12    public static void main(String[] args) {  
13        TestClass tc = new TestClass();  
14        int x = tc.multiply(2);  
15        tc.add(x);  
16        System.out.println("Result: " + tc.x);  
17    }  
18 }
```

Memory

TestClass object
int x: 15

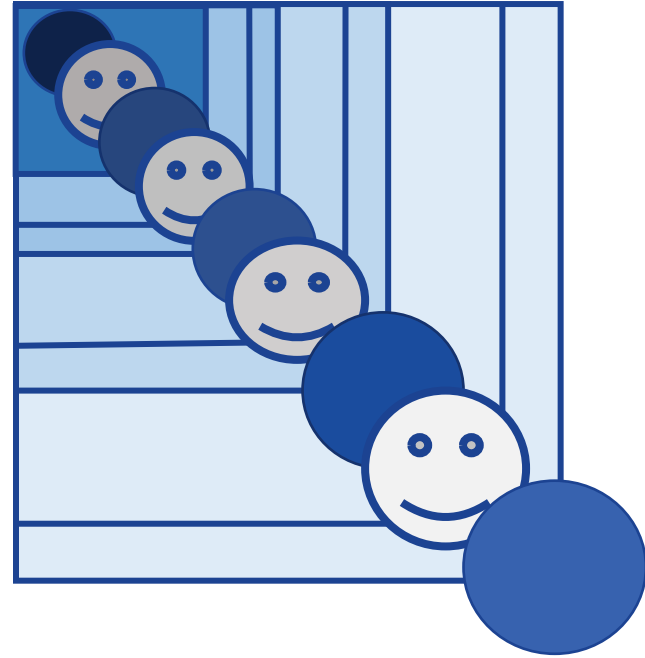
main()
TestClass tc: ●
int x: 10

Recursion

- A recursive method returns a call to itself until a base case is reached

```
methodX(param) {  
    if(..) {  
        methodX(newParam);  
    }  
    ..  
}
```

- Method attributes stored in call stack: deep recursion can lead to excessive memory consumption



Simple recursion example

```
1 public class SimpleRecursion {
2     public static void recursiveLoop(int i, int max){
3         System.out.print(i + " ");
4         if(i < max) {
5             recursiveLoop(i + 1, max);
6         }
7     }
8     public static void main(String[] args){
9         System.out.println();
10        recursiveLoop(1,4);
11    }
12 }
```

```
$ java SimpleRecursion
```

Call stack

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1
```

Call stack

recursiveLoop() [0]
int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {
2     public static void recursiveLoop(int i, int max){
3         System.out.print(i + " ");
4         if(i < max) {
5             recursiveLoop(i + 1, max);
6         }
7     }
8     public static void main(String[] args){
9         System.out.println();
10        recursiveLoop(1,4);
11    }
12 }
```

```
$ java SimpleRecursion
1
```

Call stack

recursiveLoop() [0]
int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {
2     public static void recursiveLoop(int i, int max){
3         System.out.print(i + " ");
4         if(i < max) {
5             recursiveLoop(i + 1, max);
6         }
7     }
8     public static void main(String[] args){
9         System.out.println();
10        recursiveLoop(1,4);
11    }
12 }
```

```
$ java SimpleRecursion
1 2
```

Call stack

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {
2     public static void recursiveLoop(int i, int max){
3         System.out.print(i + " ");
4         if(i < max) {
5             recursiveLoop(i + 1, max);
6         }
7     }
8     public static void main(String[] args){
9         System.out.println();
10        recursiveLoop(1,4);
11    }
12 }
```

```
$ java SimpleRecursion
1 2
```

Call stack

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1 2 3
```

Call stack

recursiveLoop() [2]

int i: 3; int max: 4

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1 2 3
```

Call stack

recursiveLoop() [2]

int i: 3; int max: 4

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1 2 3 4
```

Call stack

recursiveLoop() [3]

int i: 4; int max: 4

recursiveLoop() [2]

int i: 3; int max: 4

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {
2     public static void recursiveLoop(int i, int max){
3         System.out.print(i + " ");
4         if(i < max) {
5             recursiveLoop(i + 1, max);
6         }
7     }
8     public static void main(String[] args){
9         System.out.println();
10        recursiveLoop(1,4);
11    }
12 }
```

```
$ java SimpleRecursion
1 2 3 4
```

Call stack

recursiveLoop() [2]

int i: 3; int max: 4

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {
2     public static void recursiveLoop(int i, int max){
3         System.out.print(i + " ");
4         if(i < max) {
5             recursiveLoop(i + 1, max);
6         }
7     }
8     public static void main(String[] args){
9         System.out.println();
10        recursiveLoop(1,4);
11    }
12 }
```

```
$ java SimpleRecursion
1 2 3 4
```

Call stack

recursiveLoop() [1]

int i: 2; int max: 4

recursiveLoop() [0]

int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1 2 3 4
```

Call stack

recursiveLoop() [0]
int i: 1; int max: 4

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1 2 3 4
```

Call stack

main()

Simple recursion example

```
1 public class SimpleRecursion {  
2     public static void recursiveLoop(int i, int max){  
3         System.out.print(i + " ");  
4         if(i < max) {  
5             recursiveLoop(i + 1, max);  
6         }  
7     }  
8     public static void main(String[] args){  
9         System.out.println();  
10        recursiveLoop(1,4);  
11    }  
12 }
```

```
$ java SimpleRecursion  
1 2 3 4 $
```

Call stack

Caveats of recursion

- Mistakes like omitting the base case or writing the recursion step incorrectly can cause *infinite recursion*
- Recursive programs may result in exponential method calls
- Each recursive method can be re-written using loops
- Use recursive methods only if the problem is naturally recursive (i.e., to improve understanding) or there are performance benefits, or you need to impress in a job interview 😊

Fibonacci number example

- Fibonacci sequence: Each number is a sum of two preceding numbers, starting from 0 and 1
- Generating a Fibonacci sequence is a naturally recursive problem

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	...
0	1	1	2	3	5	8	13	21	34	55	...

Fibonacci by recursion example

```
1 public class FibonacciRecursion {
2     public static int fibonacci(int f1,int f2,int cnt) {
3         if(cnt == 2) {
4             return f1+f2;
5         } else {
6             return fibonacci(f2, f1+f2, cnt-1);
7         }
8     }
9     public static void main(String[] args) {
10         System.out.println("Result: " + fibonacci(0,1,19));
11     }
12 }
```

```
$ java FibonacciRecursion
Result: 4181
```

Summary

- Testing and dependency management are essential parts of software development cycle
 - Proper testing helps to catch the bugs at the early phase;
 - Dependency management is important to avoid broken dependencies
 - In Java development, Maven is a commonly used tool for software project management, including testing and dependency management
- Basic memory concepts in Java introduced
 - Reference type variables, pass-by-value and pass-by-reference
 - Call stack and recursion: recursion is efficient to implement algorithms that are recursive by nature, but has also caveats

Questions, comments?