Reflection in Java

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

What is reflection in Java?

- Reflection – API that allows modifying class, interface, enum, or method behavior at run-time
  - *java.lang.reflect* package
    - *import java.lang.reflect.*;*
- Reflection also allows accessing structure information (attributes and methods) at run-time
- Reflection allows instantiating an object, invoke methods, and change fields at run-time
- Ability of program code to inspect other code

Why would we use reflection?

- Extensibility
  - E.g. when your program uses external classes
- Class browsers
- IDE applications and other development tools
- Debugging
- Unit testing

Disadvantages of using reflection

- Performance overhead
  - JVM has to perform dynamic type resolution, which takes extra resources
- Security restrictions
  - Reflection requires sufficient run-time permissions
- Potential side effects
  - Reflection code is allowed operations that regular code cannot do
    - E.g. changing values of private variables
- High maintenance
  - Reflection code is hard to understand and debug
- Issues with the code most often cannot be found at compile time

Type inquiries

- Sometimes it is necessary to inquire about the type of variable we are dealing with
  - Use *instanceof()* method to check the type
- If the object is null and *instanceof()* is used then it returns false
- Returns true if the object is a subtype of a supertype

Example: type inquiry

```
interface Animal
{
    public abstract void eat();
}

class Dog implements Animal
{
    public void eat(){System.out.println("Dog is eating");}
}

public class Test
{
    public static void main(String args[])
    {
        Dog d = new Dog();

        if(d instanceof Animal){System.out.println("This object is an Animal");}
        if (d instanceof Dog){System.out.println("This object is a Dog");}
    }
}
```

Output:

```
This object is an Animal
This object is a Dog
```

Notes about type *instanceof()*

- Cannot test for a type with which there is no relationship
  - E.g. Circle is a Shape but cannot ask if Circle object is a Triangle, even if Triangle is a Shape
    - Compile error
- Every class in Java implicitly inherits from class *Object*
  - *instanceof()* evaluation of *Object* type will return true

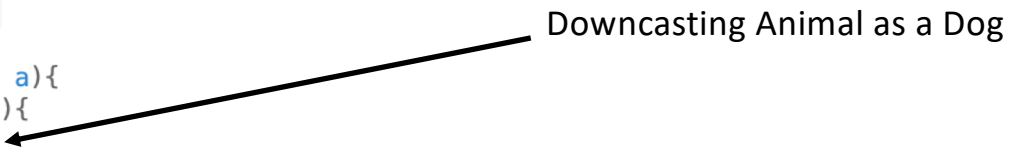
```
class Dog
{
    public void eat(){System.out.println("Dog is eating");}
}

public class test
{
    public static void main(String args[])
    {
        Dog d = new Dog();

        if(d instanceof Object){System.out.println("d is an Object type");}
    }
}
```

Concept of downcasting

- Referring to a child class as its parent class
  - E.g. *Fruit f = new Apple();*
- Improper downcasting causes run-time errors
  - Allowed by the compiler when there is a chance it might succeed at run-time
- Often useful when a parent type is passed in as a parameter
- Useful when want to allow functionality for some children of the parent type
  - Use *instanceof()* to check for type

Example: downcasting Animal object as Dog

```
interface Animal
{
    public abstract void eat();
}

class Dog implements Animal
{
    public void bark(Animal a){
        if(a instanceof Dog){
            Dog d = (Dog)a;
            System.out.println("Bark bark");
        }
    }

    public void eat(){System.out.println("Dog is eating");}
}

public class Test
{
    public static void main(String args[])
    {
        Dog d = new Dog();
        d.bark(d);
    }
}
```



Example 2: downcasting Animal object as a Dog

```
class Animal
{
}

class Dog extends Animal
{
    public static void bark(Animal a){
        Dog d = (Dog)a;
        System.out.println("Bark bark");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Animal d = new Dog();
        Dog.bark(d);
    }
}
```

Downcasting Animal object as Dog

Dog is an Animal so this is ok

Example 2: downcasting Animal object as a Dog

```java
class Animal
{
}

class Dog extends Animal
{
    public static void bark(Animal a){
        Dog d = (Dog)a;
        System.out.println("Bark bark");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Animal d = new Dog();
        Dog.bark(d);
    }
}
```

If we did this:

```java
Animal d = new Animal();
Dog.bark(d);
```

We would get a run-tim error:

```
Exception in thread "main" java.lang.ClassCastException: Animal cannot be cast to Dog
        at Dog.bark(test.java:1054)
        at test.main(test.java:1064)
```

Example: Printable type

```
interface Printable{}
class A implements Printable{
public void a(){System.out.println("a method");}
}
class B implements Printable{
public void b(){System.out.println("b method");}
}
```

```
class Call{
void invoke(Printable p){//upcasting
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
}
```

```
class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}
```

Output: `b method`

https://www.javatpoint.com

Obtain class information with *getClass()*

- Every class in Java implicitly is a subtype of Object
- *java.lang.Object.getClass()* allows retrieving the type of object and other metadata about the class at runtime

Example: retrieving type of object at runtime

```
interface Animal
{
    abstract void eat();
}

class Dog implements Animal
{
    public void eat(){
        System.out.println("I am eating");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Dog d = new Dog();
        System.out.println("I am "+d.getClass());
    }
}
```

Output:

```
I am class Dog
```

Example: retrieving type of object at runtime

```
interface Animal
{
    abstract void eat();
}

class Dog implements Animal
{
    public void eat(){
        System.out.println("I am eating");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Animal d = new Dog();
        System.out.println("I am "+d.getClass());
    }
}
```

We changed Dog to Animal here

Output is still:

```
I am class Dog
```

Example: retrieving object type from within the class methods

```
class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        Class c = getClass();
        return("<"+c.getName()+">: firstName = " + firstName + ", lastName = " + lastName);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Person p = new Person("John", "Smith");
        System.out.println(p.toString());
    }
}
```

We override Object's toString() method

Equivalent to *this.getClass()*

Java reflection *forName()* method

- *java.lang.Class.forName()* allows retrieving class object associated with a class or interface with a given name
  - *public static Class<?> forName(String name, boolean initialize, ClassLoader loader) throws ClassNotFoundException*
- The class name has to be fully classified
- Can initialize or not, using this method
- Class loader argument can override default system provided class loader

Example: obtaining class object using *forName()*

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        try{
            Class employeeClass = Class.forName("Employee");
            System.out.println(employeeClass.getName());
        }catch(ClassNotFoundException e){
            System.out.println(e);
        }
    }
}
```

Methods vs. constructors in Java reflection

- Method and Constructor classes extend *java.lang.reflect.Executable* and share a lot of inherited functionality

Example: listing class methods and constructors at run-time

Same class definition as in previous example

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        System.out.println("\nEmployee class methods:");
        Method[] employeeMethods = employeeClass.getMethods();
        for(Method method : employeeMethods){
            System.out.println(method.getName());
        }

        System.out.println("\nEmployee class declared methods:");
        Method[] employeeDeclaredMethods = employeeClass.getDeclaredMethods();
        for(Method method : employeeDeclaredMethods){
            System.out.println(method.getName());
        }

        System.out.println("\nEmployee class declared constructors:");
        Constructor[] employeeDeclaredConstructors = employeeClass.getDeclaredConstructors();
        for(Constructor method : employeeDeclaredConstructors){
            System.out.println(method.getName());
        }
    }
}
```

first, create an object of type Class from my Employee object

- *getDeclaredMethods()* lists only methods declared in the class definition, *getMethods()* lists all methods available to the class, including inherited methods

Example: listing class methods and constructors at run-time cont'd, output of the program

```
My employee: Janet Donovan (101) at Company2

Employee class methods:
toString
getName
setName
getID
setID
getEmployer
setEmployer
wait
wait
wait
equals
hashCode
getClass
notify
notifyAll

Employee class declared methods:
toString
getName
setName
getID
setID
getEmployer
setEmployer

Employee class declared constructors:
Employee
Employee
```

Example: listing class methods and constructors with argument and return types

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        System.out.println("\nEmployee class methods:");
        Method[] employeeMethods = employeeClass.getMethods();
        for(Method method : employeeMethods){
            System.out.println(method.toString());
        }

        System.out.println("\nEmployee class declared methods:");
        Method[] employeeDeclaredMethods = employeeClass.getDeclaredMethods();
        for(Method method : employeeDeclaredMethods){
            System.out.println(method.toString());
        }

        System.out.println("\nEmployee class declared constructors:");
        Constructor[] employeeDeclaredConstructors = employeeClass.getDeclaredConstructors();
        for(Constructor method : employeeDeclaredConstructors){
            System.out.println(method.toString());
        }
    }
}
```

toString() method instead of *getName()*

```
My employee: Janet Donovan (101) at Company2

Employee class methods:
public java.lang.String Employee.toString()
public java.lang.String Employee.getName()
public void Employee.setName(java.lang.String)
public int Employee.getID()
public void Employee.setID(int)
public java.lang.String Employee.getEmployer()
public void Employee.setEmployer(java.lang.String)
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()

Employee class declared methods:
public java.lang.String Employee.toString()
public java.lang.String Employee.getName()
public void Employee.setName(java.lang.String)
public int Employee.getID()
public void Employee.setID(int)
public java.lang.String Employee.getEmployer()
public void Employee.setEmployer(java.lang.String)

Employee class declared constructors:
Employee()
Employee(java.lang.String,int,java.lang.String)
```

Example: obtaining parameter type information

Same class definition as in previous example

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        //System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        Method[] employeeMethods = employeeClass.getDeclaredMethods();
        for(Method method : employeeMethods){
            String methodName = method.getName();
            Class[] parameters = method.getParameterTypes();

            if(parameters.length == 0){
                System.out.println(methodName+"() has 0 parameters");
            }else{
                System.out.println(methodName+"() has "+parameters.length+
                    " parameter(s) of the following types:");
            }
            for (Class p : parameters) {
                System.out.println("\t"+p.getName());
            }
        }
    }
}
```

Output:

```
toString() has 0 parameters
getName() has 0 parameters
setName() has 1 parameter(s) of the following types:
        java.lang.String
getID() has 0 parameters
getEmployer() has 0 parameters
setID() has 1 parameter(s) of the following types:
        int
setEmployer() has 1 parameter(s) of the following types:
        java.lang.String
```

Example: checking how many parameters of type String each method has

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        //System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        Method[] employeeMethods = employeeClass.getDeclaredMethods();
        for(Method method : employeeMethods){
            String methodName = method.getName();
            Class[] parameters = method.getParameterTypes();
            int strCount = 0;
            for(Class p : parameters){
                if(p.equals(String.class)){strCount++;}
            }

            System.out.println(methodName+"() has "+strCount+
                " parameters of type String");
        }
    }
}
```

This line counts how many parameter types equal to *String.class*

Output:

```
getName() has 0 parameters of type String
setName() has 1 parameters of type String
getID() has 0 parameters of type String
setID() has 0 parameters of type String
getEmployer() has 0 parameters of type String
setEmployer() has 1 parameters of type String
```

Example: listing fields in class

Same class definition as in previous example

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        System.out.println("Field names:");
        Field[] employeeFields = Employee.class.getDeclaredFields();
        for(Field field : employeeFields){
            System.out.println(field.getName());
        }

        System.out.println("\nField names and types:");
        for(Field field : employeeFields){
            System.out.println(field.toString());
        }
    }
}
```

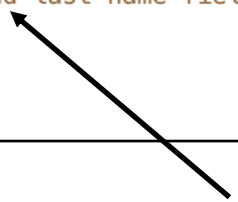
Output:

```
My employee: Janet Donovan (101) at Company2
Field names:
name
id
employer

Field names and types:
private java.lang.String Employee.name
private int Employee.id
private java.lang.String Employee.employer
```

Example: testing we found the field of interest

Same Employee class as before

```
public class Test {
    public static void main(String args[]){
        Field[] employeeFields = Employee.class.getDeclaredFields();
        for(Field field : employeeFields){
            if(field.getName().equals("name") && field.getType().equals(String.class)){
                System.out.println("Found name field");
            }
        }

        for(Field field : employeeFields){
            if(field.getName().equals("lastName") && field.getType().equals(String.class)){
                System.out.println("Found last name field");
            }
        }
    }
}
```



This line will not execute as there is no field named *lastName* in class *Employee*

Output:

```
Found name field
```

Example: invoking a method by name at run-time

Same class definition as in previous example

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

argument type of *setName()* method

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        try{
            Method setNameMethod = employeeClass.getDeclaredMethod("setName", String.class);
            setNameMethod.invoke(e1, "James Fillmore");

            System.out.println("My employee: "+e1.toString());
        }catch(NoSuchMethodException | IllegalAccessException | InvocationTargetException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

Output:

```
My employee: Janet Donovan (101) at Company2
My employee: James Fillmore (101) at Company2
```

Example: invoking a method without input arguments

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

Same class definition as in previous example

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");

        Class employeeClass = e1.getClass();

        try{
            Method toStringMethod = employeeClass.getDeclaredMethod("toString");
            System.out.println("My employee: "+(String)toStringMethod.invoke(e1));
        }catch(NoSuchMethodException | IllegalAccessException | InvocationTargetException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

return type

no input arguments into *toString()* method

Output:

```
My employee: Janet Donovan (101) at Company2
```

Example: invoking a method with input arguments and return type

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }

    public double computePay(int hours, double regularRate, double overtimeRate){
        int overtimeHours = hours - 40;
        if(overtimeHours > 0){
            return 40*regularRate + overtimeHours*overtimeRate;
        }else{
            return hours*regularRate;
        }
    }
}
```

let's add *computePay()* method to *Employee* class

Example: invoking a method with input arguments and return type (cont'd)

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");

        Class employeeClass = e1.getClass();

        try{
            Method computePayMethod = employeeClass.getDeclaredMethod("computePay", int.class, double.class, double.class);
            System.out.println("Employee pay this period = $"+(double)computePayMethod.invoke(e1, 45, 10, 20));
        }catch(NoSuchMethodException | IllegalAccessException | InvocationTargetException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

Output:

```
Employee pay this period = $500.0
```

Example: changing the values of class fields

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        try{
            Field nameField = employeeClass.getDeclaredField("name");

            System.out.println("Accessible = "+nameField.isAccessible());

            nameField.setAccessible(true);
            nameField.set(e1, "John Smith");
            System.out.println("My employee: "+e1.toString());

            System.out.println("Accessible = "+nameField.isAccessible());
            nameField.setAccessible(false);
        }catch(NoSuchFieldException | IllegalAccessException e){
            System.out.println("Warning: problems working with name field");
        }
    }
}
```

Same class definition as in previous examples

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

Output:

```
My employee: Janet Donovan (101) at Company2
Accessible = false
My employee: John Smith (101) at Company2
Accessible = true
```

Example: invoking a private class method

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }

    private String privateHelper(){
        return "additional employee information";
    }
}
```

private method

```java
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();

        try{
            Method privateHelperMethod = employeeClass.getDeclaredMethod("privateHelper");
            System.out.println("Accessible = "+privateHelperMethod.isAccessible());
            privateHelperMethod.setAccessible(true);
            System.out.println((String)privateHelperMethod.invoke(e1));
            privateHelperMethod.setAccessible(false);
        }catch(NoSuchMethodException | IllegalAccessException | InvocationTargetException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

Output:

```
Accessible = false
additional employee information
```

If we don't set accessible to true first, exception will be thrown:

```
Accessible = false
Warning: problems invoking the method
```

Example: invoking a static method

```java
import java.lang.reflect.*;
import java.util.Random;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }

    public static int generateRandomNumber(){
        Random random = new Random();
        int randomInteger = random.nextInt();
        return randomInteger;
    }
}
```

static method

```java
public class Test {
    public static void main(String args[]){
        try{
            Class<?> employeeClass = Class.forName("Employee");
            Method numberGeneratorMethod = employeeClass.getMethod("generateRandomNumber");

            boolean accessibleFlag = true;
            if(!numberGeneratorMethod.isAccessible()){
                accessibleFlag = false;
                numberGeneratorMethod.setAccessible(true);
            }

            System.out.println((int)numberGeneratorMethod.invoke(null));

            if(!accessibleFlag){
                numberGeneratorMethod.setAccessible(false);
            }
        }catch(NoSuchMethodException | IllegalAccessException |
            InvocationTargetException | ClassNotFoundException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

Output:

```
1293146805
```

Example: invoking a static method with arguments

```java
import java.lang.reflect.*;
import java.util.Random;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }

    public static int generateRandomNumber(int upperLimit){
        Random random = new Random();
        int randomInteger = random.nextInt(upperLimit);
        return randomInteger;
    }
}
```

Input argument specifies the range of random numbers

```java
public class Test {
    public static void main(String args[]){
        try{
            Class<?> employeeClass = Class.forName("Employee");
            Method numberGeneratorMethod = employeeClass.getMethod("generateRandomNumber", int.class);

            boolean accessibleFlag = true;
            if(!numberGeneratorMethod.isAccessible()){
                accessibleFlag = false;
                numberGeneratorMethod.setAccessible(true);
            }

            System.out.println((int)numberGeneratorMethod.invoke(null, 100));

            if(!accessibleFlag){
                numberGeneratorMethod.setAccessible(false);
            }
        }catch(NoSuchMethodException | IllegalAccessException |
            InvocationTargetException | ClassNotFoundException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

Need to specify input argument type(s)

input argument value

Example: obtaining class name and package information

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();
        System.out.println(employeeClass.getSimpleName());
        System.out.println(employeeClass.getName());
        System.out.println(employeeClass.getTypeName());
    }
}
```

Same class definition as in previous examples

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

- *getName()* returns fully classified name as specified in the class declaration (name used to dynamically load the class), for inner classes ”$” are used in the path, e.g. *java.lang.String* or java.lang.Runnable
- *getSimpleName()* returns loose class name/path (useful for logging and debugging purposes), e.g. *String* or *Runnable*
- *getTypeName()* returns an “informative string for this type”; similar to toString() method in many built in classes

Example: obtaining class modifier information

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();
        int employeeModifiers = employeeClass.getModifiers();
        System.out.println("Is Employee public: "+Modifier.isPublic(employeeModifiers));
        System.out.println("Is Employee public: "+Modifier.isProtected(employeeModifiers));
        System.out.println("Is Employee public: "+Modifier.isPrivate(employeeModifiers));
        System.out.println("Is Employee abstract: "+Modifier.isAbstract(employeeModifiers));
        System.out.println("Is Employee static: "+Modifier.isStatic(employeeModifiers));
        System.out.println("Is Employee modifier: "+Modifier.toString(employeeModifiers));
    }
}
```

Can inquire if the object has a particular modifier or can just get the modifier as a string

Example: get package information

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();
        Package employeePackage = employeeClass.getPackage();
        try{
            System.out.println("Employee class package: "+employeePackage.getName());
        }catch(NullPointerException e){
            System.out.println("This class is not in a package");
        }
    }
}
```

Example: getting superclass information

```java
import java.lang.reflect.*;

class Person{
    private String name;

    Person(){
        this.name = "";
    }

    Person(String name){
        this.name = name;
    }

    public String getName(){return this.name;}
    public void setName(String name){this.name = name;}
}
```

```java
class Employee extends Person{

    private int id;
    private String employer;

    Employee(){
        super();
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        super(name);
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return super.getName();}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){super.setName(name);}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return super.getName() + " (" + this.id + ") at "+this.employer;
    }
}
```

Example: getting superclass information cont'd

```
public class Test {
    public static void main(String args[]){
        System.out.println("Fields in Employee's superclass:");
        Field[] personFields = Employee.class.getSuperclass().getDeclaredFields();
        for(Field field : personFields){
            int fieldModifier = field.getModifiers();
            System.out.println(Modifier.toString(fieldModifier)+" "+field.getName());
        }

        System.out.println("\nMethods in Employee's superclass:");
        Method[] personMethods = Employee.class.getSuperclass().getDeclaredMethods();
        for(Method method : personMethods){
            System.out.println(method.getName());
        }
    }
}
```

Output:

```
Fields in Employee's superclass:
private name

Methods in Employee's superclass:
getName
setName
```

Example: invoking methods in superclass

Same Person and Employee class definitions as in previous examples

```
public class Test {
    public static void main(String args[]){
        Employee e1 = new Employee("Janet Donovan", 101, "Company2");
        System.out.println("My employee: "+e1.toString());

        Class employeeClass = e1.getClass();
        try{
            Method setNameMethod = employeeClass.getSuperclass().getMethod("setName",String.class);

            boolean accessibleFlag = true;
            if(!setNameMethod.isAccessible()){
                accessibleFlag = false;
                setNameMethod.setAccessible(true);
            }

            setNameMethod.invoke(e1, "Will Smith");
            System.out.println("My employee: "+e1.toString());

            if(!accessibleFlag){
                setNameMethod.setAccessible(false);
            }
        }catch(NoSuchMethodException | IllegalAccessException |
            InvocationTargetException e){
            System.out.println("Warning: problems invoking the method");
        }
    }
}
```

Output:

```
My employee: Janet Donovan (101) at Company2
My employee: Will Smith (101) at Company2
```

Example: listing interfaces the class implements

```java
import java.lang.reflect.*;

interface Eats{
    abstract public void eat();
}

interface Walks{
    abstract public void walk();
}

interface Sleeps{
    abstract public void sleep();
}

class Animal implements Eats, Walks, Sleeps{
    private String type;
    private String sound;

    Animal(String type, String sound){
        this.type = type;
        this.sound = sound;
    }

    public void eat(){System.out.println("I am eating");}
    public void walk(){System.out.println("I am walking");}
    public void sleep(){System.out.println("I am sleeping");}

    public String toString(){
        return "I am "+this.type+", I make sound "+this.sound;
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        Animal a1 = new Animal("cat", "meow");
        System.out.println(a1.toString());

        Class animalClass = a1.getClass();
        Class[] animalInterfaces = animalClass.getInterfaces();

        for(Class interf : animalInterfaces){
            System.out.println(interf.getName());
        }

    }
}
```

We use Class type here

Output:

```
I am cat, I make sound meow
Eats
Walks
Sleeps
```

Example: creating new class instance

Same class definition as in previous examples

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

```
public class Test {
    public static void main(String args[]){
        try{
            Employee employeeInstance = Employee.class.newInstance();
            employeeInstance.setName("Joe Smith");
            employeeInstance.setID(200);
            employeeInstance.setEmployer("Big company");
            System.out.println("My employee: "+employeeInstance.toString());
        }catch(InstantiationException | IllegalAccessException e){
            System.out.println(e);
        }
    }
}
```

Output:

```
My employee: Joe Smith (200) at Big company
```

Class.newInstance() internally calls *Constructor.newInstance()*

Example: creating new class instance using *forName()*

Same class definition as in previous example

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

creates Employee instance from class name

```
public class Test {
    public static void main(String args[]){
        try{
            Employee employeeInstance = (Employee) Class.forName("Employee").newInstance();
            employeeInstance.setName("Joe Smith");
            employeeInstance.setID(200);
            employeeInstance.setEmployer("Big company");
            System.out.println("My employee: "+employeeInstance.toString());
        }catch(ClassNotFoundException | InstantiationException | IllegalAccessException e){
            System.out.println(e);
        }
    }
}
```

Output:

```
My employee: Joe Smith (200) at Big company
```

Example: creating new instance with *Constructor.newInstance()* method

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    public Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    public Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

Employee constructors

```java
public class Test {
    public static void main(String args[]){
        try{
            Constructor<Employee> employeeConstructor = Employee.class.getConstructor();
            Employee employeeInstance = employeeConstructor.newInstance();
        }catch(NoSuchMethodException | InstantiationException |
            IllegalAccessException | InvocationTargetException e){
            System.out.println(e);
        }
    }
}
```

Call *Constructor.newInstance()* method

Create a constructor object for a constructor with no parameters