

# JAVA

Runtime Type Information and Reflection. Dynamic  
Type Loading, Inversion of control

Date: 5-December-2019

Course: Programming Languages

Lecturer: Dejan Mitov

# The Need for Reflection?

Apps do not always control types used

- *Common in advance app designs*
- *Common in tools and frameworks*

Often dynamically load types

- *Type not known at compile time*
- *There's no type-specific source code*

Requires special runtime type handling

- *Examine types at runtime*
- *Dynamically execute & access members*

# Goals

---

Reflection overview

---

Java type representation

---

Accessing a type's Class instance

---

Accessing information about type

---

Accessing type member information

---

Interaction with object instances

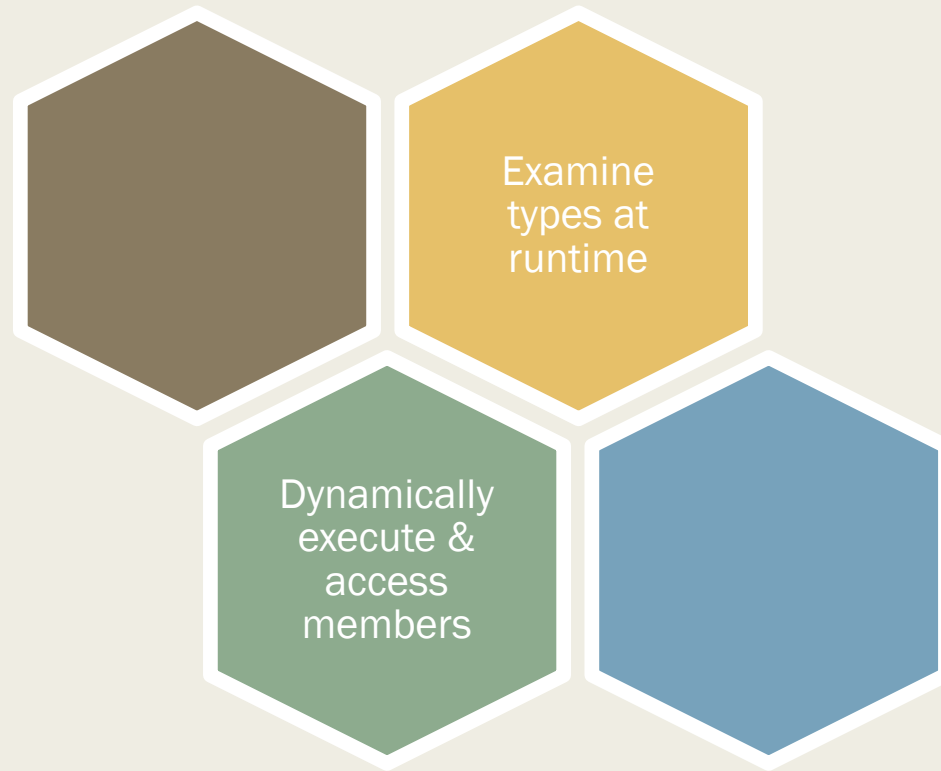
---

Create type instances

---

Dynamic type loading

# Reflection core capabilities



# Runtime Examination

Can fully  
examine  
objects  
at  
runtime

*Type, Base types*  
*Interfaces implemented*  
*Members*

Variety  
of uses

*Determine a type's capabilities*  
*Tools development*

- Type inspector
- Schema generator

# Dynamic Execution and Access

## Can access full capability of type

- *Construct instances*
- *Access fields*
- *Call methods*

## Variety of uses

- *Configurable application designs*
  - Specifics tasks externally controlled
- *Inversion of control application designs*
  - App provides fundamental behavior
  - Classes added to specialize the behavior

# Type as a Type!

Type is the foundation of any app solution

- *We use types to model biz issues*
- *We use types to model tech issues*

Java uses type to model type issues

- *Fundamental type is the Class class*
  - Each type has a Class instance
  - Describes the type in detail

## Instance of *Class* class

simpleName

BankAccount

fields

id

balance

constructors

BankAccount

BankAccount

methods

getId

getBalance

deposit

withdrawal

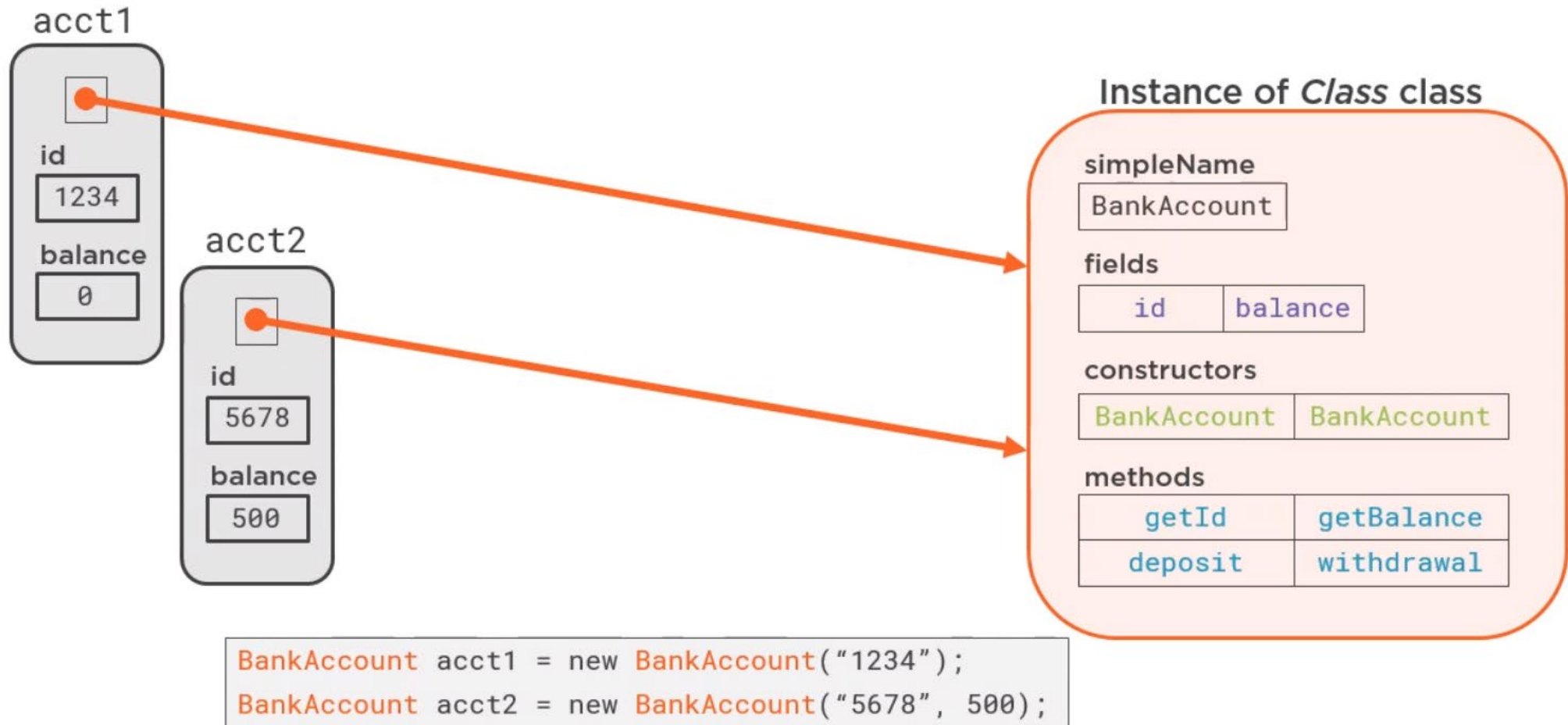
```
public class BankAccount {  
    private final String id;  
    private int balance = 0;  
  
    public BankAccount(String id) {...}  
    public BankAccount(String id, int balance) {...}  
  
    public String getId() {...}  
    public synchronized int getBalance() {...}  
    public synchronized void deposit(int amount) {...}  
    public synchronized void withdrawal(int amount) {...}  
}
```

```
BankAccount acct1 = new BankAccount("1234");  
BankAccount acct2 = new BankAccount("1234", 500);
```

# CLASS DECLARATION



# Class Declaration



# Accessing a Type's Class Instance

- Accessing a type's Class instance
  - From a type *reference*
    - Call getClass method
  - From *string name*
    - Call Class.forName static method
      - Pass fully qualified type name
  - From *type literal*
    - Use typename.class

# Class Instance from Type Reference

```
BankAccount acct = new BankAccount("1234");  
doWork(acct);
```

```
void doWork(object obj) {  
    Class<?> c = obj.getClass();  
    showName(c);  
}
```

```
void showName(Class<?> theClass) {  
    System.out.println(theClass.getSimpleName());  
}
```

BankAccount

# Class Instance from String Name or Type Literal

```
Class<?> c = Class.forName("com.jwhh.finance.BankAccount");  
showName(c);
```

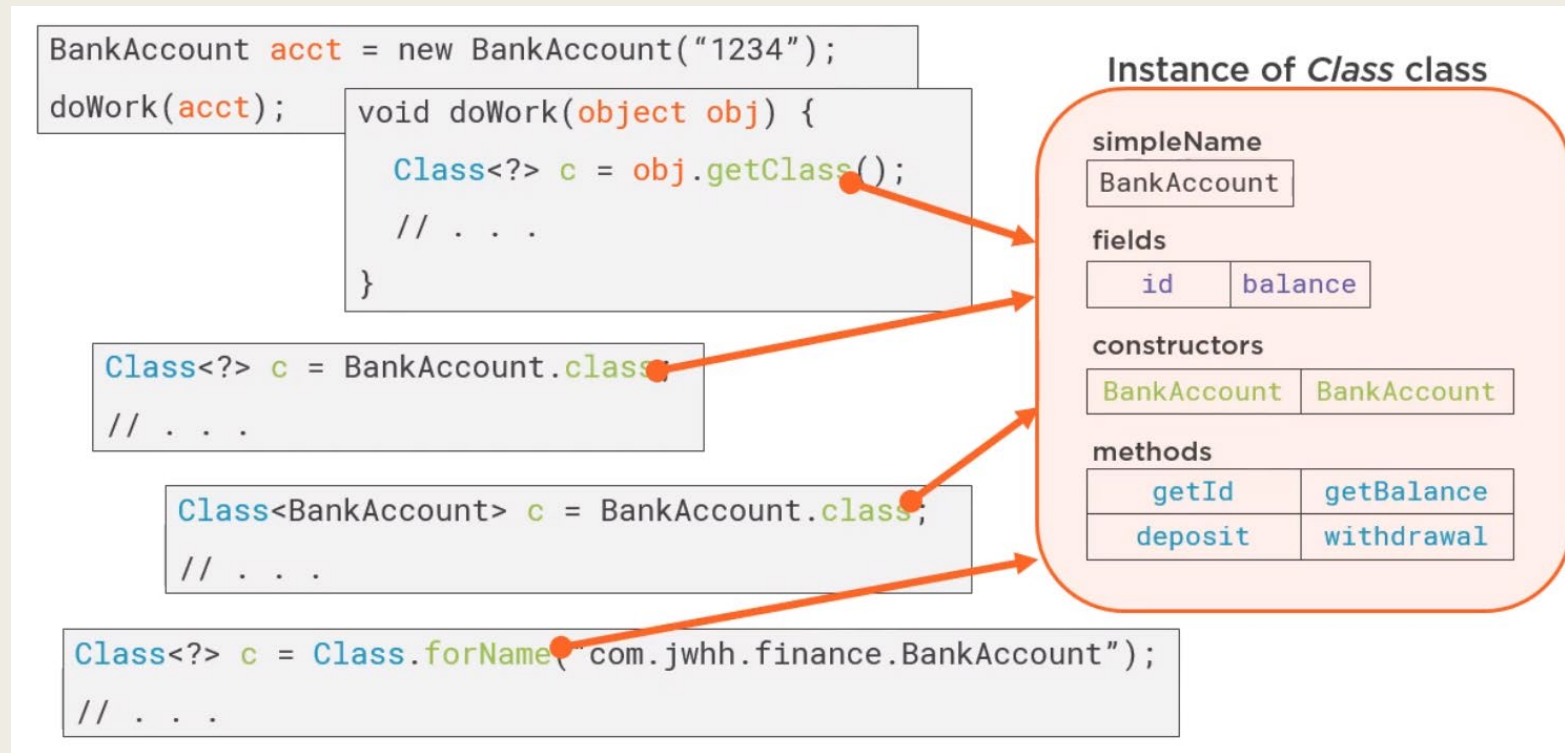
```
Class<?> c = BankAccount.class;  
showName(c);
```

```
Class<BankAccount> c = BankAccount.class;  
showName(c);
```

```
void showName(Class<?> theClass) {  
    System.out.println(theClass.getSimpleName());  
}
```

```
BankAccount  
BankAccount  
BankAccount
```

# Type Has One Class Instance



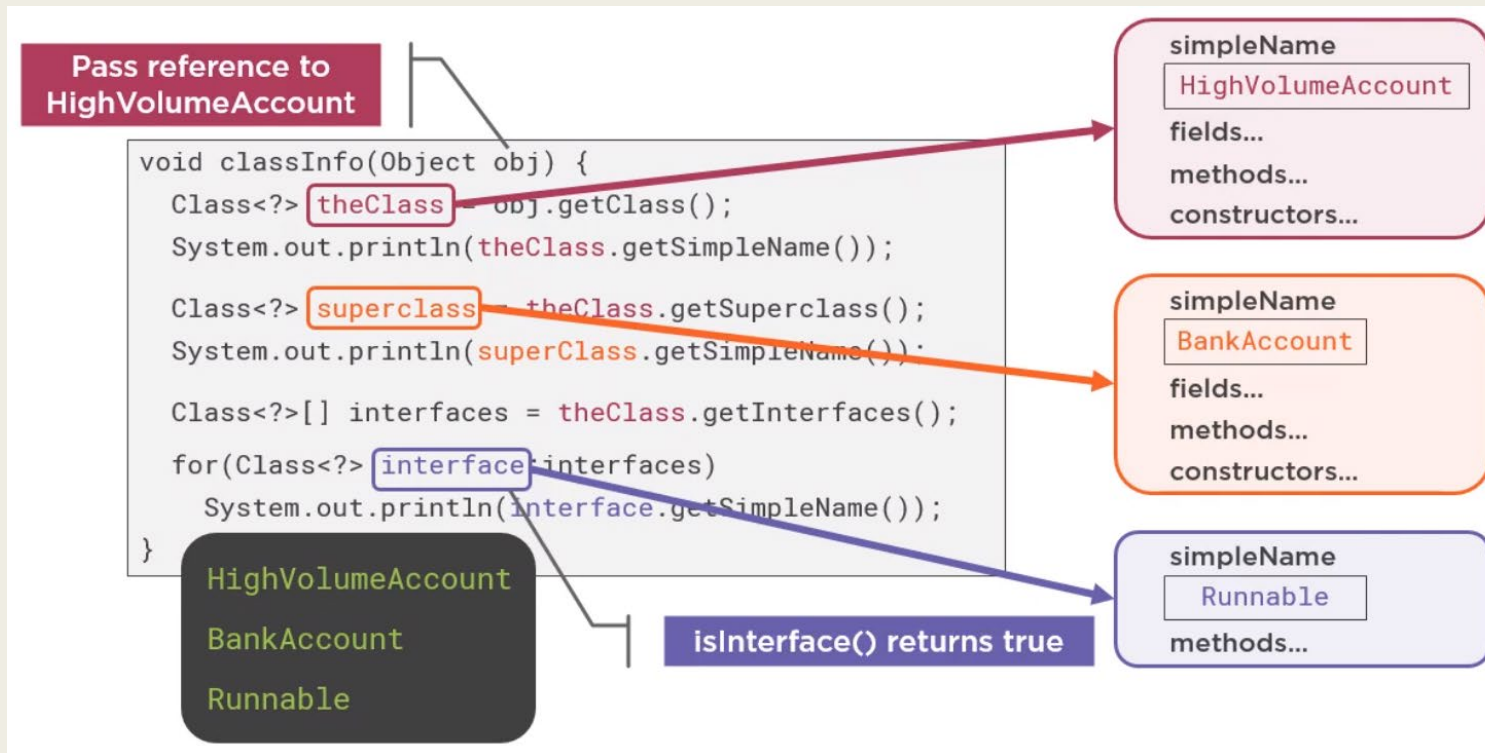
# Accessing Type Information

- Every aspect of the type is knowable
  - *Superclass*
  - *Implemented interfaces*
  - *Modifiers*
  - *Members*

# Accessing Type information

```
public final class HighVolumeAccount extends BankAccount implements Runnable {  
  
    public HighVolumeAccount(String id) { super(id); }  
    public HighVolumeAccount(String id, int balance) { super(id, balance); }  
  
    private int[] readDailyDeposits() {...}  
    private int[] readDailyWithdrawals() {...}  
  
    public String run() {  
  
        for(int depositAmt:readDailyDeposits())  
            deposit(depositAmt);  
  
        for(int withdrawalAmt:readDailyWithdrawals())  
            withdrawal(withdrawalAmt);  
    }  
}
```

# Accessing Superclass and interfaces





# Type access modifiers

- Retrieve type access modifiers
  - *Use `getModifiers`*
  - *Returned as a single `int` value*
    - Each modifier is a separate bit
- Use Modifier class to interpret modifiers
  - *Provides static fields for bit comparisons*
    - Requires use of bitwise and/or
  - *Provides static helper methods*
    - Each checks for specific modifier

# Type Access Modifiers

```
public final class HighVolumeAccount extends BankAccount implements Runnable {  
  
    public HighVolumeAccount(String id) { super(id); }  
    public HighVolumeAccount(String id, int balance) { super(id, balance); }  
  
    private int[] readDailyDeposits() {...}  
    private int[] readDailyWithdrawals(int amount) {...}  
  
    public String run() {  
  
        for(int depositAmt:readDailyDeposits())  
            deposit(depositAmt);  
  
        for(int withdrawalAmt:readDailyWithdrawals())  
            withdrawal(withdrawalAmt);  
    }  
}
```

# Retrieving Type Access Modifiers

```
void typeModifiers(Object obj) {  
    Class<?> theClass = obj.getClass();  
    int modifiers = theClass.getModifiers();  
  
    if((modifiers & Modifier.FINAL) > 0)  
        System.out.println("bitwise check - final");  
  
    if(Modifier.isFinal(modifiers))  
        System.out.println("method check - final");  
  
    if(Modifier.isPrivate(modifiers))  
        System.out.println("method check - private");  
    else if(Modifier.isProtected(modifiers))  
        System.out.println("method check - protected");  
    else if(Modifier.isPublic(modifiers))  
        System.out.println("method check - public");  
}
```

bitwise check - final  
method check - final  
Method check - public

# Type Describe Type Members



Name  
Type

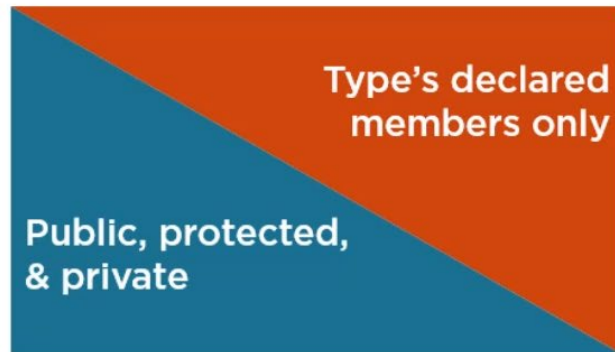


Name  
Return type  
Parameter types

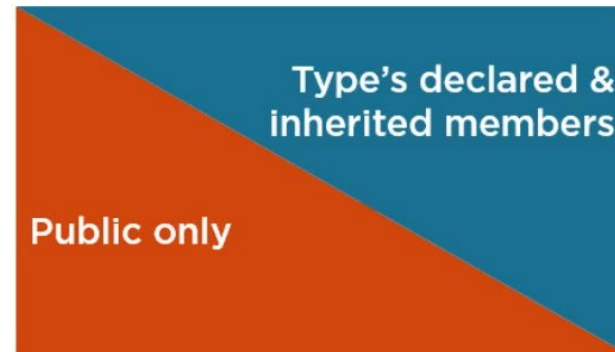


Name

# Accessing Type Members



getDeclaredFields  
getDeclaredMethods  
getDeclaredConstructors



getFields  
getMethods  
getConstructors

Limited to public  
on current type



# Accessing Field Information

```
public class BankAccount {  
    private final String id;  
    private int balance = 0;  
    // other members elided  
}
```

```
void fieldInfo(Object obj) {  
    Class<?> theClass = obj.getClass();  
    Field[] fields = theClass.getFields();  
    displayFields(fields);  
  
    Field[] declaredFields = theClass.getDeclaredFields();  
    displayFields(declaredFields);  
}
```

```
void displayFields(Field[] arr) {  
    for(Field f:arr)  
        System.out.println(f.getName() + " : " + f.getType());  
}
```

id : String

# Accessing Method Information


```
public class BankAccount {  
    public String getId() {...}  
    public synchronized int getBalance() {...}  
    public synchronized void deposit(int amount) {...}  
    public synchronized void withdrawal(int amount) {...}  
  
    // other members elided  
}
```

```
public final class HighVolumeAccount extends BankAccount implements Runnable {  
    private int[] readDailyDeposits() {...}  
    private int[] readDailyWithdrawals(int amount) {...}  
    public void run() {...}  
  
    // other members elided  
}
```

# Accessing Method Information

Pass reference to  
HighVolumeAccount

```
void methodInfo(Object obj) {  
    Class<?> theClass = obj.getClass();  
  
    Method[] methods = theClass.getMethods();  
    for(Method m:methods)  
        // . . .  
  
    Method[] declMethods = theClass.getDeclaredMethods();  
    for(Method m:declMethods)  
        // . . .  
}
```



⋮
toString
getClass
equals
getId
getBalance
deposit
withdrawal
run

readDailyDeposits
readDailyWithdrawals
run



# Excluding Object Class Methods

```
void methodInfo2(Object obj) {  
    Class<?> theClass = obj.getClass();  
  
    Method[] methods = theClass.getMethods();  
    for(Method m:methods) {  
        if(m.getDeclaredClass() != Object.class)  
            System.out.println(m.getName());  
    }  
}
```

getId
getBalance
deposit
withdrawal
run

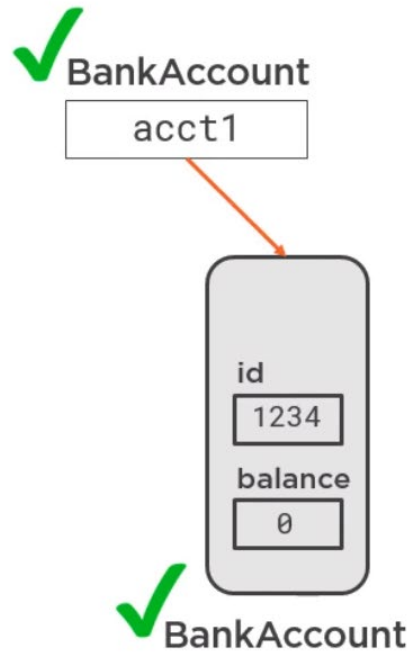
# More About Members

- Can request individual member by signature
  - *getFields*
    - Pass name
  - *getMethod*
    - Pass name plus parameter types
  - *getConstructor*
    - Pass parameter types
- Members have access modifiers
  - Use *getModifiers*
  - Interpret with *Modifier* class

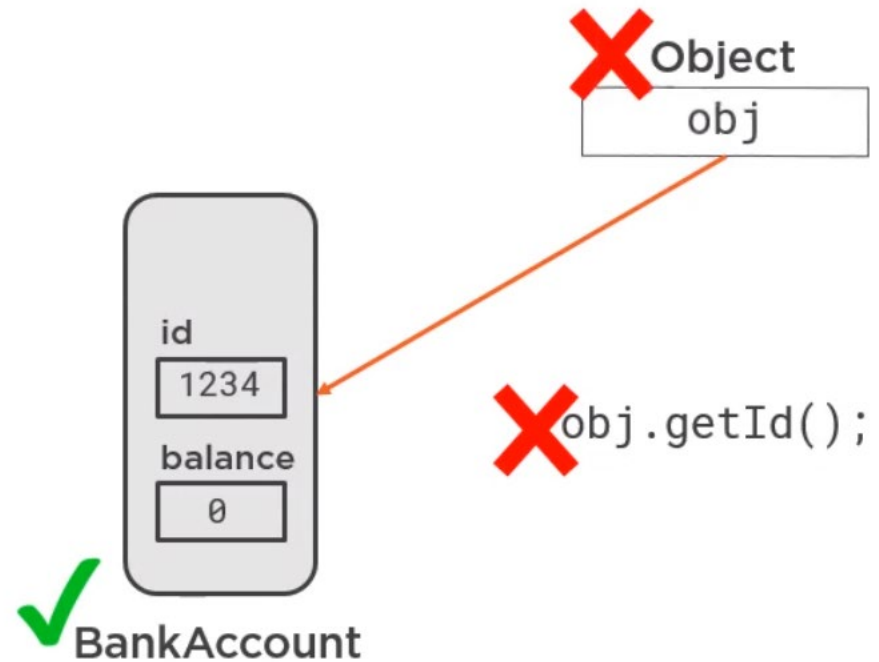
# Interaction with Object Instances

- Interacting with object instances
  - *Reflection not limited to describing types*
  - *Can access and invoke members*

# Member Access



`acct1.getId();`



# Method Access with Reflection

```
BankAccount acct1 = new BankAccount("1234");  
callGetId(acct1);
```

Result: 1234

```
void callGetId(Object obj) {  
    try {  
        Class<?> theClass = obj.getClass();  
        Method m = theClass.getMethod("getId");  
        Object result = m.invoke(obj);  
        System.out.println("Result: " + result);  
    } catch (Exception e) {  
        // . . .  
    }  
}
```

# Method Access with Reflection

```
BankAccount acct1 = new BankAccount("1234", 500);  
callDeposit(acct1, 50);  
System.out.println("Balance: " + acct1.getBalance());
```

Balance: 550

```
void callDeposit(Object obj, int amt) {  
    try {  
        Class<?> theClass = obj.getClass();  
        Method m = theClass.getMethod("deposit", int.class);  
        m.invoke(obj, amt);  
    } catch (Exception e) {  
        // . . .  
    }  
}
```

# Instance Creation with Reflection

- Objects can be created with reflection
  - *Constructors can be executed*
    - Use Constructor newInstance method
    - Returns a reference to new instance
  - *Simplified handling for no-arg constructor*
    - No need to access constructor directly
    - Use Class newInstance method

# Work Targets

```
public class BankAccount {  
  
    public String getId() {...}  
    public synchronized int getBalance() {...}  
    public synchronized void deposit(int amount) {...}  
    public synchronized void withdrawal(int amount) {...}  
  
    // other members elided  
}
```

```
public final class HighVolumeAccount extends BankAccount implements Runnable {  
  
    private int[] readDailyDeposits() {...}  
    private int[] readDailyWithdrawals() {...}  
    public void run() {...}  
  
    // other members elided  
}
```



# Worker

```
public class AccountWorker implements Runnable {  
    BankAccount ba;  
    HighVolumeAccount hva;  
    public AccountWorker(BankAccount ba) { ... }  
    public AccountWorker(HighVolumeAccount hva) { ... }  
    public void doWork() {  
        Thread t = new Thread(hva != null ? hva : this);  
        t.start();  
    }  
}
```

```
    public void run() {  
        char txType = // read tx type  
        int amt = // read tx amount  
        if(txType == 'w')  
            ba.withdrawal(amt);  
        else  
            ba.deposit(amt);  
    }  
}
```

# Work Dispatch System Invocation

```
void startWork(String workerTypeName, Object workerTarget) {  
    try {  
        Class<?> workerType = Class.forName(workerTypeName);  
        Class<?> targetType = workerTarget.getClass();  
        Constructor c = workerType.getConstructor(targetType);  
        Object worker = c.newInstance(workerTarget);  
        Method doWork = workerType.getMethod("doWork");  
        doWork.invoke(worker);  
    } catch (Exception e) {  
        // . . .  
    }  
}
```

```
BankAccount acct1 = new BankAccount();  
startWork("com.jwhh.utils.AccountWorker", acct1);
```

# Instance Creation with Reflection

- Updated flexible work dispatch system
  - *Core requirements same as before*
- Method to start work
  - *Accepts same 2 arguments as before*
- Worker type requirements
  - *Has a no-argument constructor*
  - *Implements TaskWorker interface*

```
public interface TaskWorker {  
    void setTarget(Object target);  
    void doWork();  
}
```

# Worker Implementing Interface

```
public class AccountWorker implements Runnable, TaskWorker {  
    BankAccount ba;  
  
    public void setTarget(object target) {  
        if(BankAccount.class.isInstance(target))  
            ba = (BankAccount)target;  
        else  
            throw new IllegalArgumentException( ... ) ;  
    }  
  
    public void doWork() {  
        Thread t = new Thread(  
            HighVolumeAccount.class.isInstance(ba) ? (HighVolumeAccount)ba : this);  
  
        t.start();  
    }  
  
    public void run() {...}  
}
```

# Work Dispatch System Interface Invocation

```
void startWork(String workerTypeName, Object workerTarget) {  
    try {  
        Class<?> workerType = Class.forName(workerTypeName);  
        TaskWorker worker = (TaskWorker) workerType.newInstance();  
        worker.setTarget(workerTarget);  
        worker.doWork();  
    } catch (Exception e) {  
        // . . .  
    }  
}
```

```
BankAccount acct1 = new BankAccount();  
startWork("com.jwhh.utils.AccountWorker", acct1);
```

# Summary

Every type is represented by Instance of Class

- Each type has exactly one instance

Accessing type's Class instance

- From type reference
  - Call getClass
- From string name
  - Call Class.forName
- From type literal
  - Use typename.class

# Summary

## All aspects of type knowable

- *Superclass and interfaces*
- *Fields, methods, and constructors*
- *Access modifiers*

## Working with modifiers

- *Returned as an int value*
- *Use Modifiers class to interpret*
  - Provides static fields for bit values
  - Provides helper methods

# Summary

## Interacting with object instances

- *Reflection not limited to describing types*
- *Can access and invoke members*

## Objects can be created with reflection

- *Constructors can be executed*
  - Use Constructor newInstance method
- *Simplified handling for no-arg constructor*
  - Use Class newInstance method





QUESTIONS?