

Runtime Type Information and Reflection



Jim Wilson

Mobile Solutions Developer & Architect

@hedgehogjim | jwhh.com

Overview



Reflection overview

Java type representation

Accessing a type's Class instance

Accessing information about a type

Accessing type member information

Interacting with object instances

Creating type instances

Dynamic type loading



Reflection



Core capabilities of reflection

- Examine types at runtime
- Dynamically execute & access members



The Need for Reflection



Apps do not always control types used

- Common in advanced app designs
- Common in tools and frameworks

Often dynamically load types

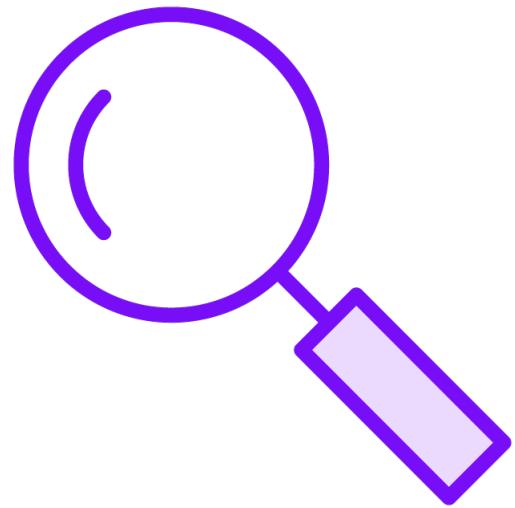
- Type not known at compile time
- There's no type-specific source code

Requires special runtime type handling

- Examine types at runtime
- Dynamically execute & access members



A Quick Look at the Basics



Can fully examine objects at runtime

Type, Base types

Interfaces implemented

Members



Variety of uses

Determine a type's capabilities

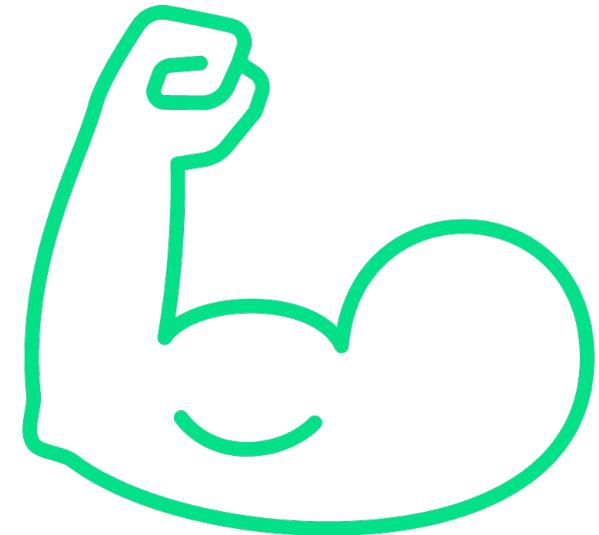
Tools development

Type inspector/browser

Schema generation

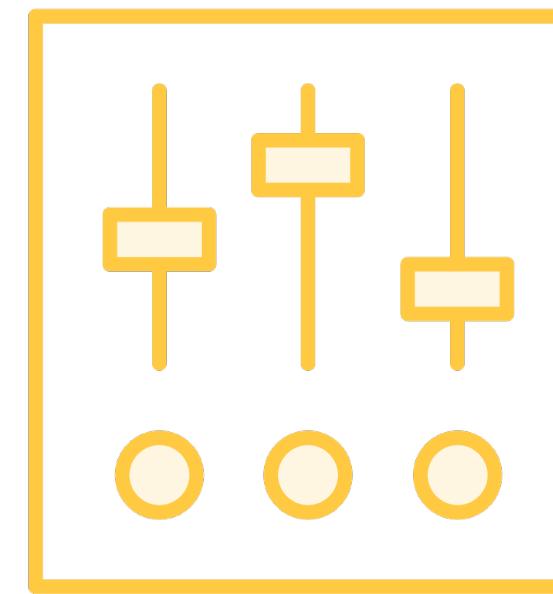


Dynamic Execution and Access



**Full access to type
capability**

Construct instances
Access fields
Call methods



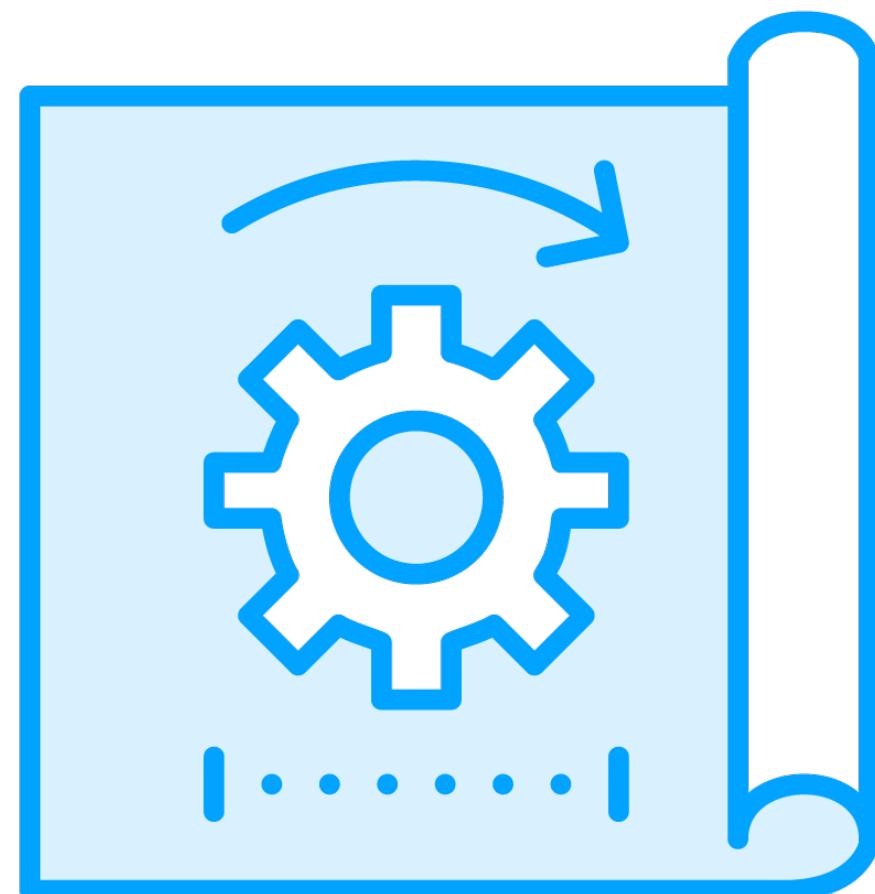
**Configurable
app designs**
**Specific tasks externally
controlled**



**Inversion of control
app designs**
**App provides fundamental
behavior**
**Classes add specialized
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 types to model type issues

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



Class Declaration

```
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("5678", 500);
```



Class Representation

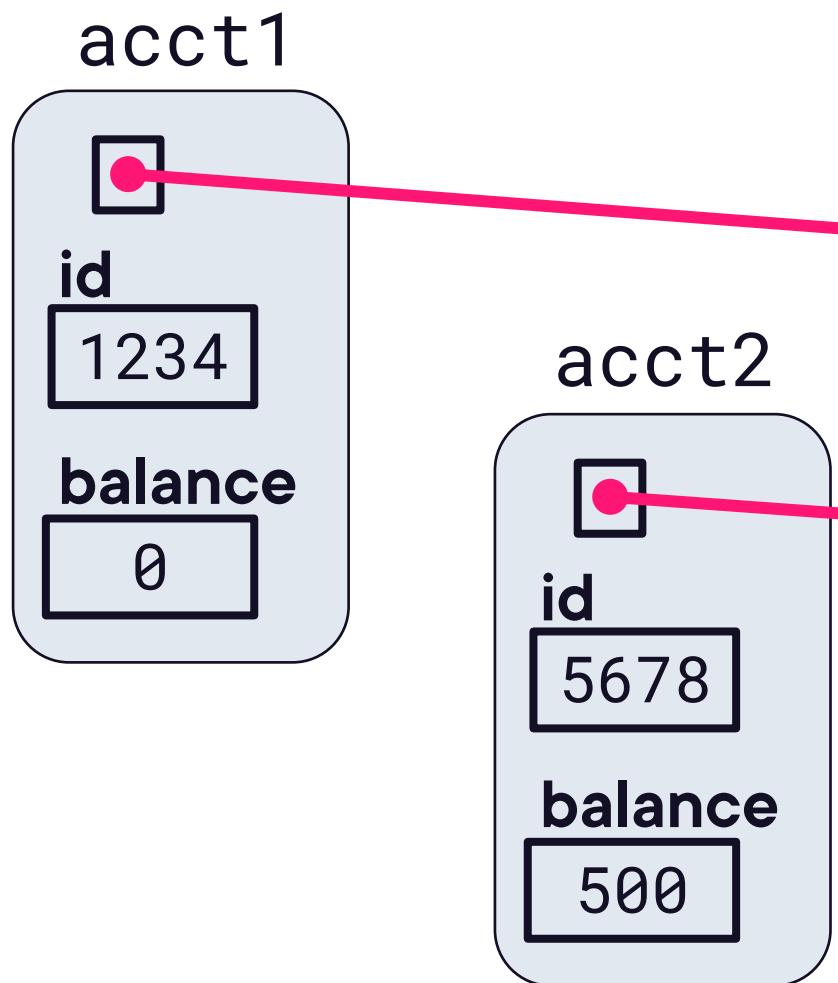
```
public class BankAccount {  
    private final String id;  
    private int balance = 0;  
    public BankAccount(String id) {...}  
    public BankAccount(String id, int balance) {...}  
    public String getId() {...}  
    public synchronized int getBalance() {...}  
    public synchronized void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

Instance of Class class

simpleName	BankAccount
fields	
id	balance
constructors	
BankAccount	BankAccount
methods	
getId	getBalance
deposit	withdrawal



Class Creation



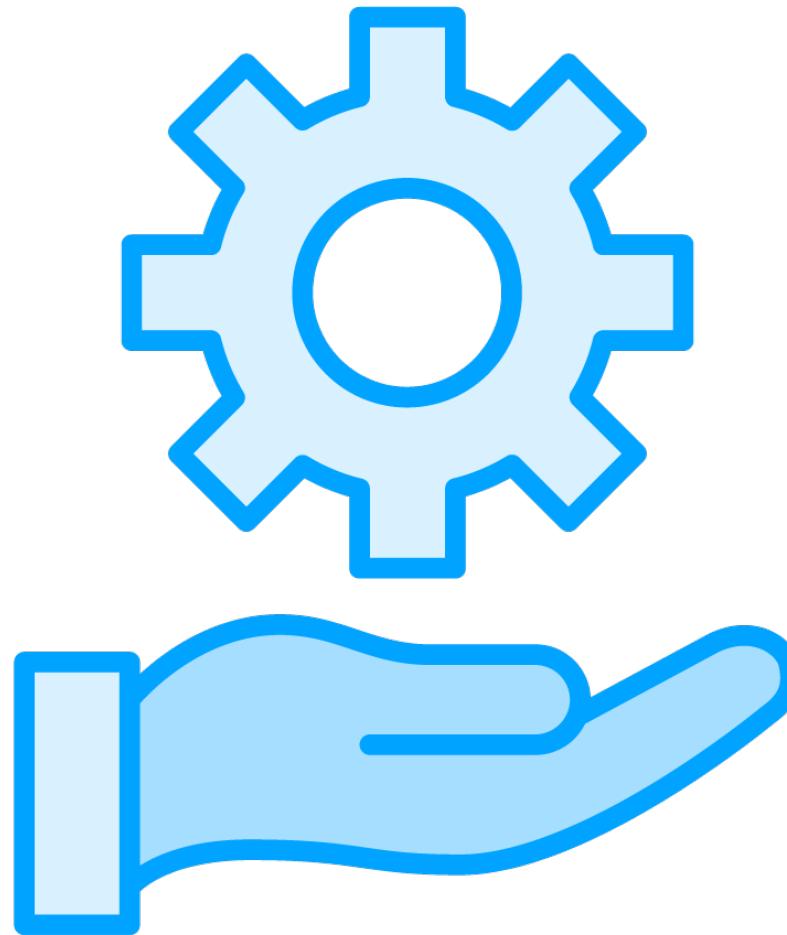
Instance of Class class

simpleName	BankAccount
fields	<code>id</code> <code>balance</code>
constructors	BankAccount BankAccount
methods	<code>getId</code> <code>getBalance</code>
	<code>deposit</code> <code>withdrawal</code>

```
BankAccount acct1 = new BankAccount("1234");
BankAccount acct2 = new BankAccount("5678", 500);
```



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

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

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

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

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

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

Instance of *Class* class

simpleName

BankAccount

fields

id	balance
----	---------

constructors

BankAccount	BankAccount
-------------	-------------

methods

getId	getBalance
deposit	withdrawal



Accessing Type Information



Every aspect of a type is knowable

- Superclass
- Implemented interfaces
- Modifiers
- Members



Accessing Type Information

```
public final class HighVolumeAccount {
```

```
}
```



Accessing Type Information

```
public final class HighVolumeAccount extends BankAccount {  
    public HighVolumeAccount(String id) { super(id); }  
    public HighVolumeAccount(String id, int balance) { super(id, balance); }  
    private int[] readDailyDeposits() {...}  
    private int[] readDailyWithdrawals() {...}  
}
```



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

Pass reference to
HighVolumeAccount

```
void classInfo(Object obj) {  
  
    Class<?> theClass = obj.getClass();  
    System.out.println(theClass.getSimpleName());  
    theClass.getSuperclass();  
    System.out.println(superClass.getSimpleName());  
    theClass.getInterfaces();  
    for(Class<?> interface:interfaces)  
        System.out.println(interface.getSimpleName());  
}
```

HighVolumeAccount
BankAccount
Runnable

simpleName

HighVolumeAccount

fields...

methods...

constructors...

simpleName

BankAccount

fields...

methods...

constructors...

isInterface() returns true

simpleName

Runnable

methods...



Type Access Modifiers



Retrieving 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() {...}  
    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();  
    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



Types to Describe Type Members

Field

Method

Constructor

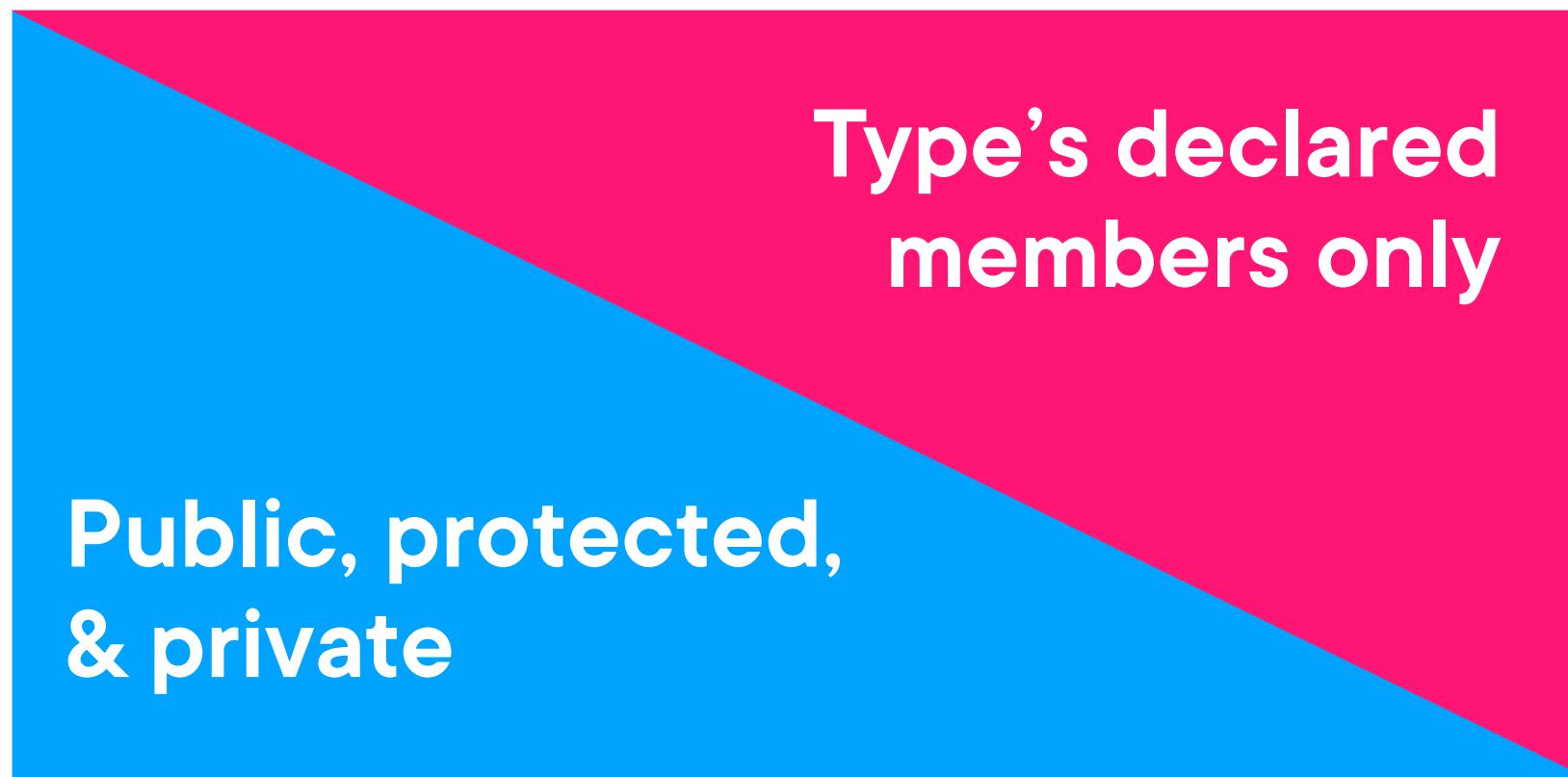
Name
Type

Name
Return type
Parameter types

Name
Parameter types

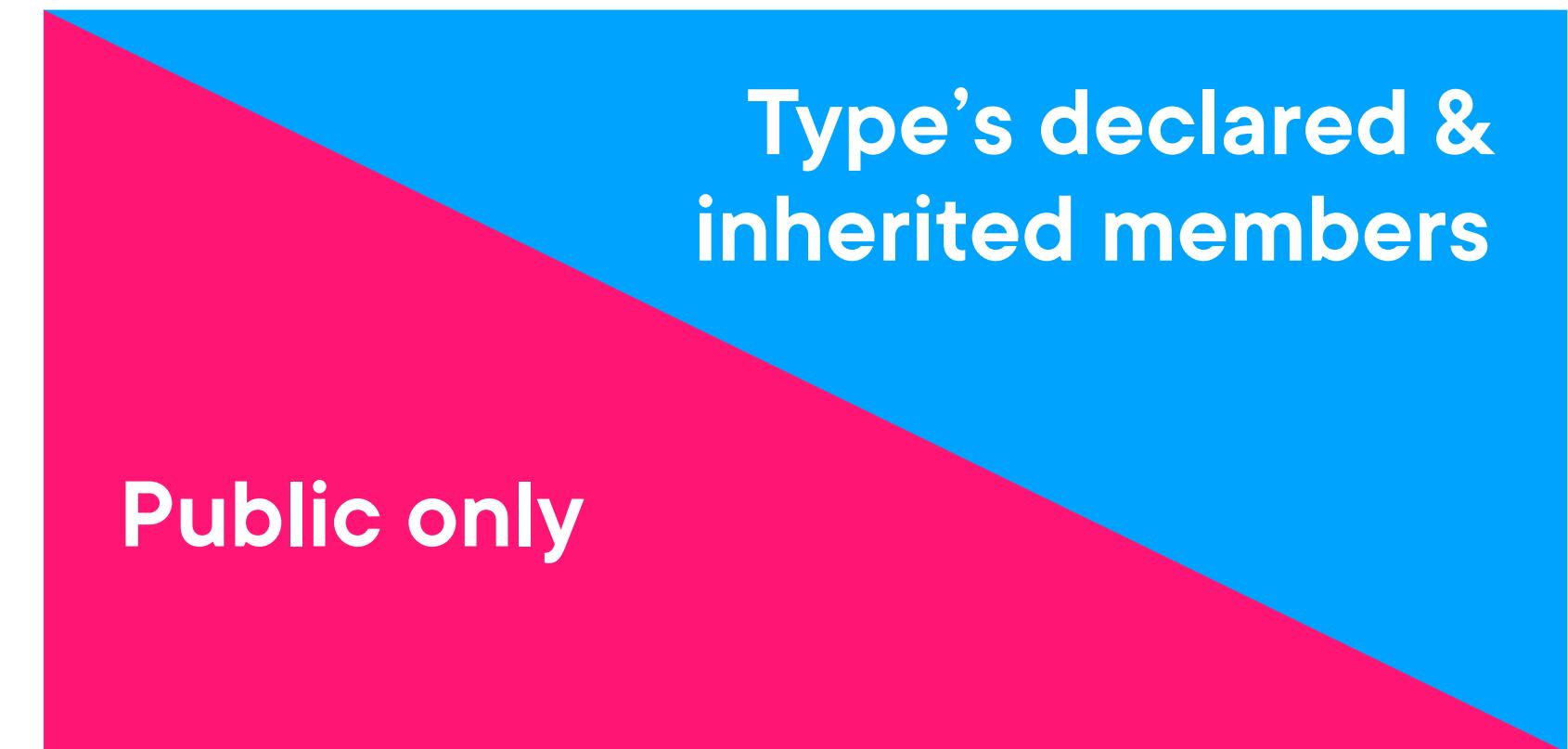


Accessing Type Members



getDeclaredFields
getDeclaredMethods
getDeclaredConstructors

Limited to public
on current type



getFields
getMethods
getConstructors



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
balance : int



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  
}
```

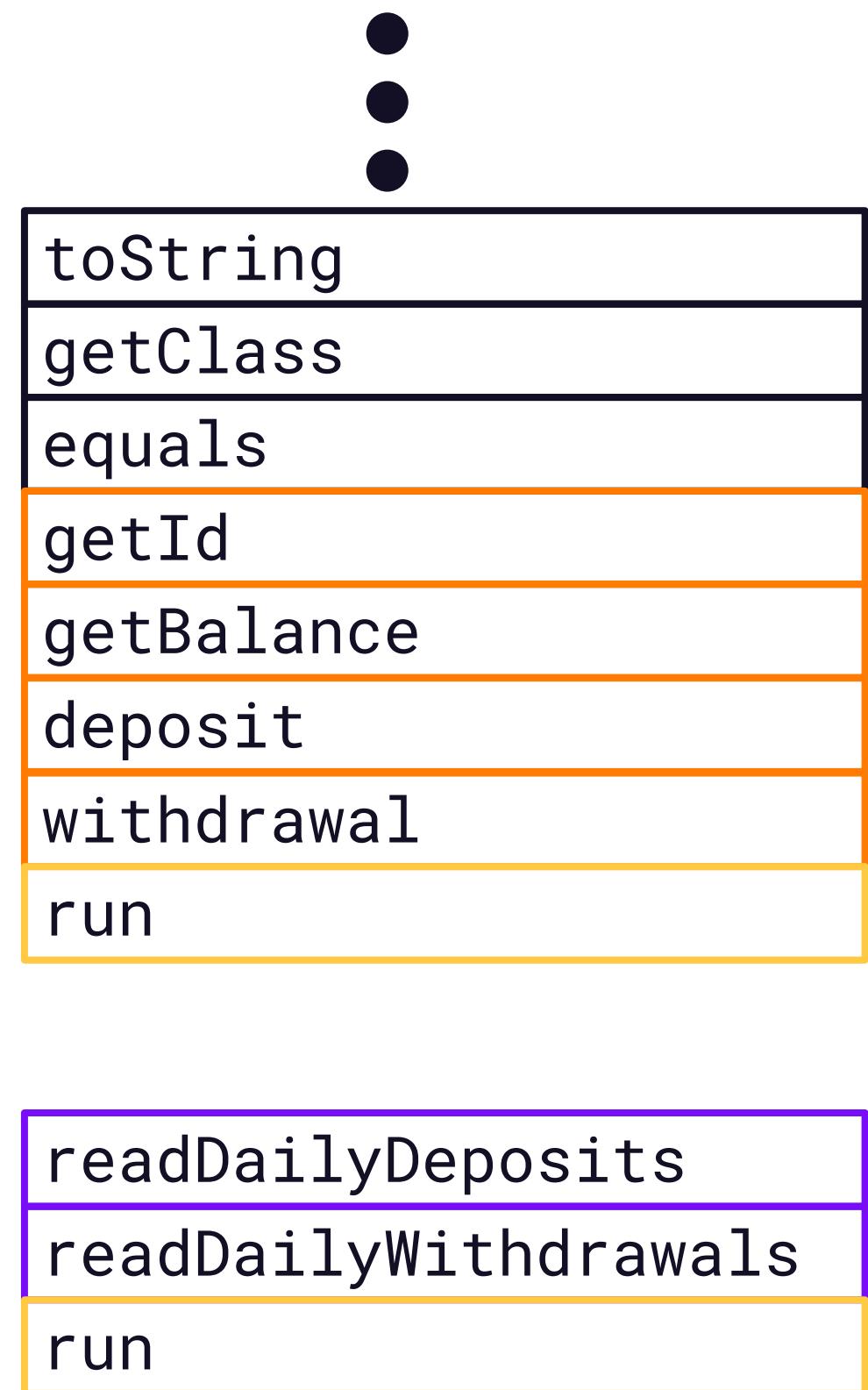
```
public final class HighVolumeAccount extends BankAccount implements Runnable {  
    private int[] readDailyDeposits() {...}  
    private int[] readDailyWithdrawals() {...}  
    public void run() {...}  
    // other members elided  
}
```



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)  
        // . . .  
}
```



Excluding Object Class Methods

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

getId
getBalance
deposit
withdrawal
run



Can request individual member by signature

More About Members

`getField`

- Pass name

`getMethod`

- Pass name plus parameter types

`getConstructor`

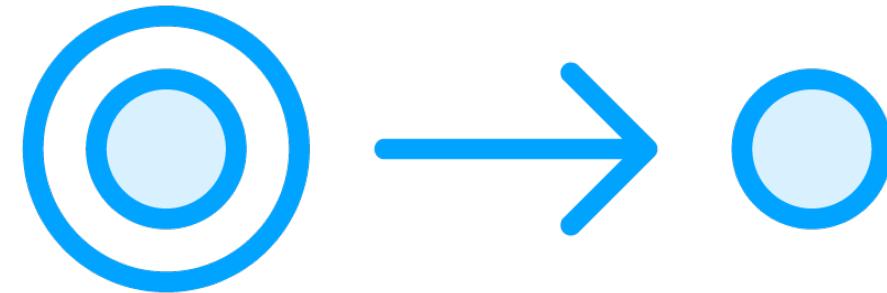
- Pass parameter types

Members have access modifiers

- Use `getModifiers`
- Interpret with `Modifier` class



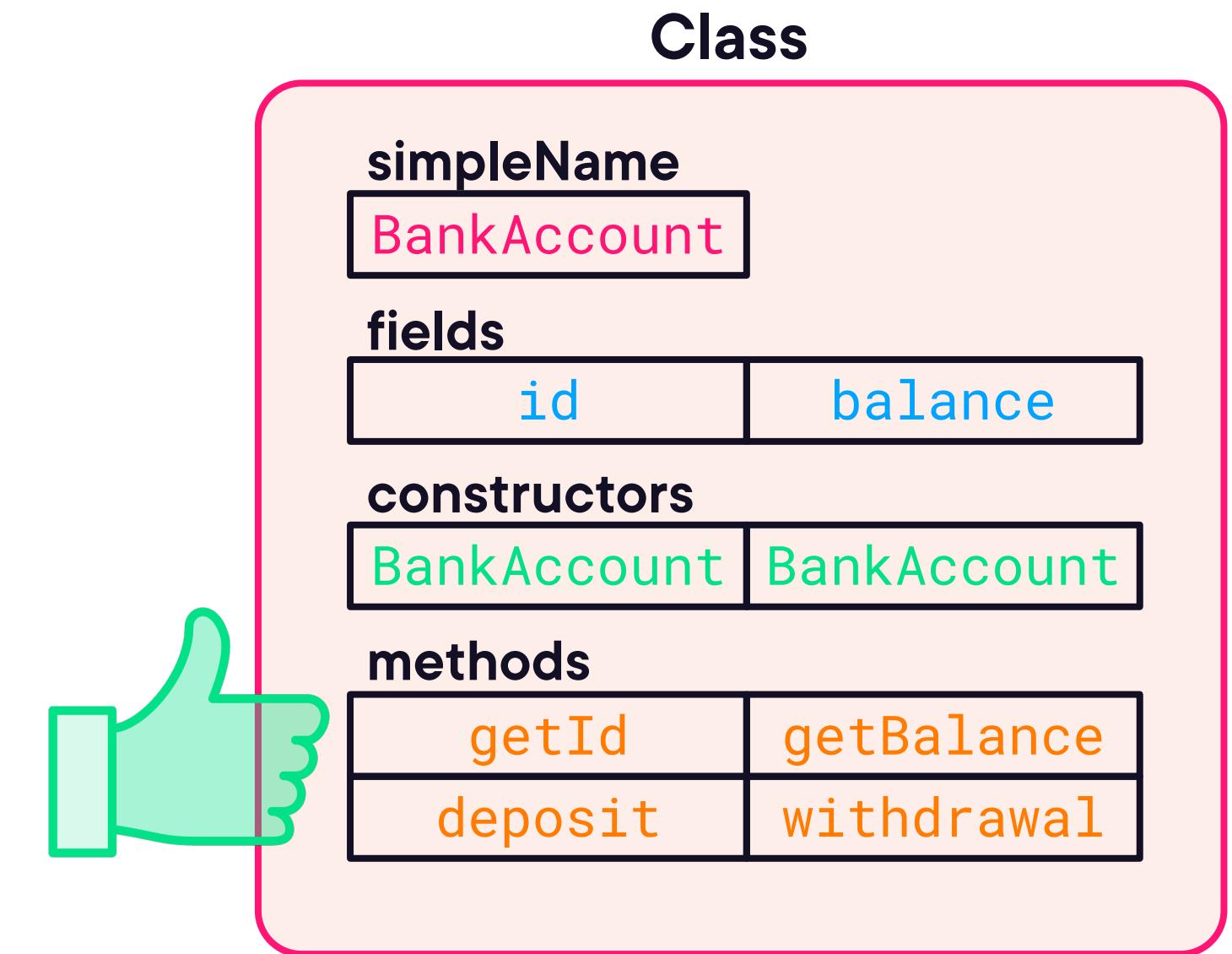
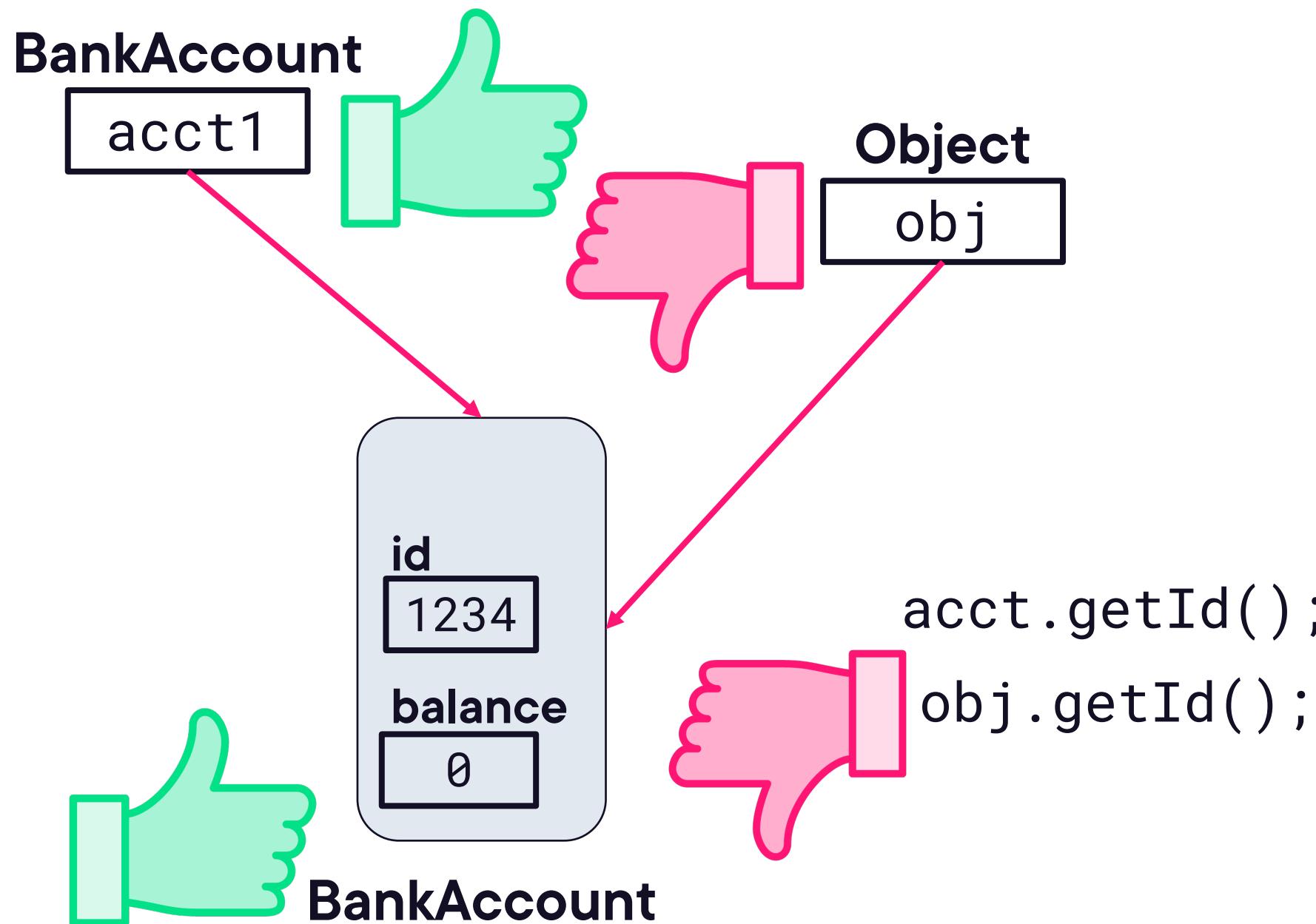
Interacting with Object Instances



Reflection not limited to describing types
Can access and invoke members



Member Access



Method Access with Reflection

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

Result: 1234

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



Method Access with Reflection

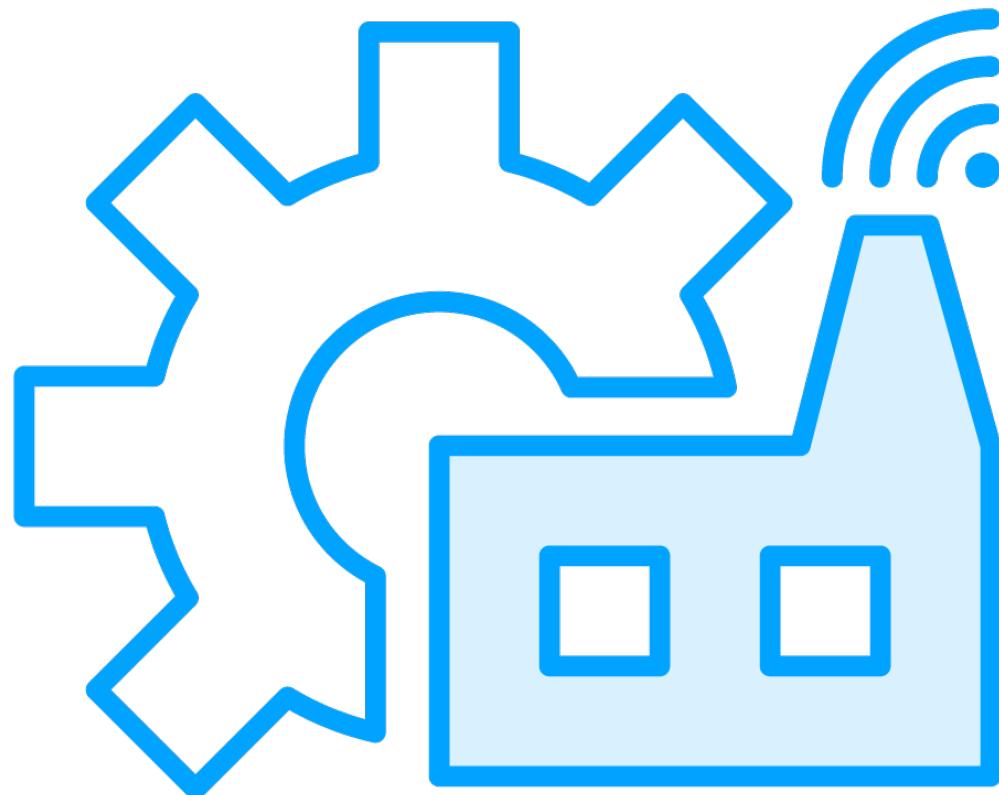
```
BankAccount acct1 = new BankAccount("1234", 500);
callDeposit(acct1, 50);
System.out.println("Balance: " + acct1.getBalance());
```

Balance: 550

```
void callDeposit(Object obj int amt) {
    try {
        Class<?> theClass = obj.getClass();
        Method m = theClass.getMethod(
            m.invoke(obj)
        } 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



Instance Creation with Reflection



Flexible work dispatch system

- Executes worker classes against targets
- Can use any worker in classpath



Instance Creation with Reflection



Method to start work accepts 2 arguments

- Name of worker type
 - Received as a String reference
- Target of work
 - Received as Object reference

Worker type requirements

- Constructor that accepts target type
- A doWork method that takes no args



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(  
            () -> {  
                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.forName(workerTypeName);  
        workerTarget.getClass();  
        workerType.getConstructor();  
        c.newInstance();  
        workerType.getMethod("doWork");  
        doWork.invoke(worker);  
    } catch(Exception e) {  
        // . . .  
    }  
}
```

```
BankAccount acct1 = new BankAccount();  
startWork( );
```



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 {  
    BankAccount ba;  
  
    public void setTarget(object target) {  
        if(BankAccount  
            ba = (BankAccount)target;  
        else  
            throw new IllegalArgumentException( ... ) ;  
    }  
  
    public void doWork() {  
        Thread t = new Thread(  
            HighVolumeAccount.class.isInstance(ba)  
        t.start();  
    }  
  
    public void run() {...}  
}
```



Work Dispatch System Interface Invocation

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

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



Summary



Every type 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 types knowable

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

Working with modifiers

- Returned as an int value

Use Modifier 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

