CSE 331

Reflection

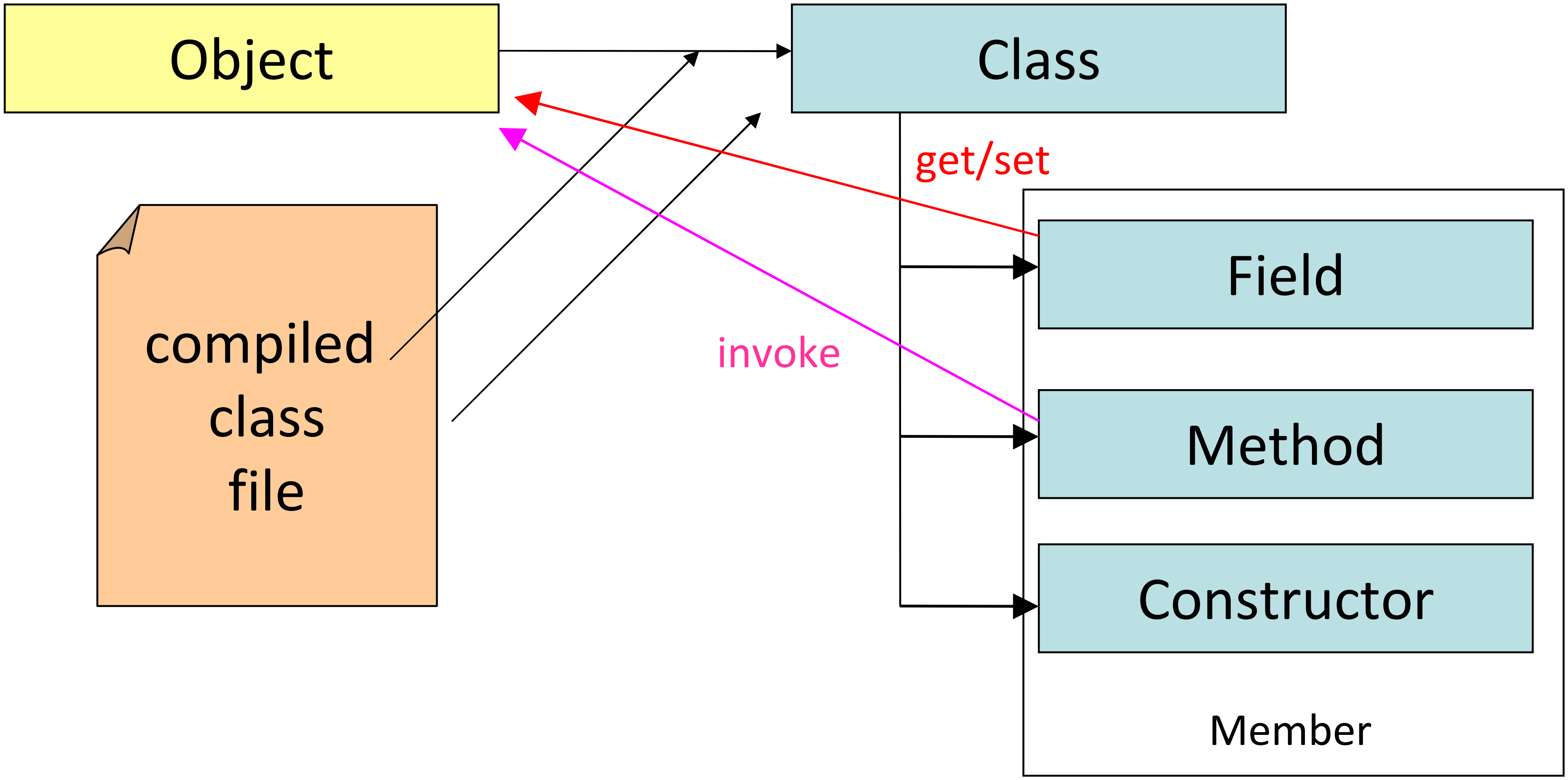
slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Reflection

- **reflection**: A process by which a program can examine its own types and structure at runtime.
  - sometimes called *run-time type inference (RTTI)*
  - `import java.lang.reflect.*;`
- Using reflection, you can:
  - Convert strings and others into classes and objects at runtime.
  - Ask detailed questions in code about the abilities of a type.
  - Dynamically compile, load, and add classes to a running program.
  - Pass function pointers (via `Method` objects)
- Reflection is used internally by many Java technologies including IDEs/compilers, debuggers, serialization, Java Beans, RMI, ...

The Class class

- An object of type `Class` represents information about a Java class.
  - Its fields, methods, constructors, superclass, interfaces, etc.
  - A gateway to the rest of Java's reflection system.



Accessing a Class object

- Ways to get a `Class` object:
  - *If you have an object:* Every object has a `getClass` method to return the `Class` object that corresponds to that object's type.
    - `Class<Point> pointClass = p.getClass();`
  - *If you don't have an object, but know the class's name at compile time:* Every class has a static field named `class` storing its `Class` object.
    - `Class<Point> pointClass = Point.class;`
  - *If you don't know the type until given its name as a string at runtime:* The static method `Class.forName(String)` will return the `Class` object for a given type; pass it a full class name.
    - `Class<?> clazz = Class.forName("java.awt.Point");`

Class class methods

method	description
getConstructor(**params**) getConstructors()	objects representing this class's constructors
getField(**name**) getFields()	objects representing this class's fields
getInterfaces()	interfaces implemented by this class
getMethod(**name, params**) getMethods()	objects representing this class's methods
getModifiers()	whether the class is public, static, etc.
getName()	full name of this class, as a string
getPackage()	object representing this class's package
newInstance()	constructs a new object of this type (if the type has a parameterless constructor)
toString()	string matching the class's header

Class class methods 2

method	description

`getAnnotation(`**class**`)`
`getAnnotations()`
	information about annotations on the class

`getResource(`**name**`)`
`getResourceAsStream(`**name**`)`
	resource-loading features

`getSuperclass()`	a `Class` object for this type's superclass
`getSimpleName()`	class name without package name
`getTypeParameters()`	all generic type params in this class

`isAnnotation()`
`isAnnotationPresent(`**type**`)`
	information about annotation types

`isAnonymousClass()`
`isArray(), isEnum()`
`isInterface(),isPrimitive()`
	testing whether the class fits into one of the given categories of types

`isAssignableFrom(`**class**`)`	whether this class is the same as or a supertype of the given class parameter
`getDeclaredFields(), ...`	fields/methods/etc. declared in this file

Reflection example

- Print all the methods and fields in the `Point` class:

```
for (Method method : Point.class.getMethods()) {
    System.out.println("a method: " + method);
}

for (Field field : Point.class.getFields()) {
    System.out.println("a field: " + field);
}
```

Primitives and arrays

- Primitive types and `void` are represented by `Class` constants:

constant	alternate form	primitive
`Integer.TYPE`	`int.class`	`int`
`Double.TYPE`	`double.class`	`double`
`Character.TYPE`	`char.class`	`char`
`Boolean.TYPE`	`boolean.class`	`boolean`
`Void.TYPE`	`void.class`	`void`
`...`	…	…

  - Not to be confused with `Integer.class`, `Double.class`, etc., which represent the wrapper classes `Integer`, `Double`, etc.

- Array classes are manipulated in reflection by static methods in the `Array` class (not to be confused with `java.util.Array`**`s`**).

Generic Class class

- As of Java 1.5, the Class class is generic: `Class<T>`
  - This is so that known types can be instantiated without casting.

```
Class<Point> clazz = java.awt.Point.class;
Point p = clazz.newInstance();  // no cast
```

- For unknown types or `Class.forName` calls, you get a `Class<?>` and must still cast when creating instances.

```
Class<?> clazz = Class.forName("java.awt.Point");
Point p = (Point) clazz.newInstance();  // must cast
```

Method class methods

method	description
`getDeclaringClass()`	the class that declares this method
`getExceptionTypes()`	any exceptions the method may throw
`getModifiers()`	whether the method is public, static, etc.
`getName``()`	method's name as a string
`getParameterTypes()`	info about the method's parameters
`getReturnType()`	info about the method's return type
`invoke``(`**obj**`,` **params**`)`	calls this method on given object *(*`null` *if static)*, passing given parameter values
`toString()`	string matching the method's header

Reflection example 1

- Calling various `String` methods in an Interactions pane:

```
// "abcdefg".length() => 7
> Method lengthMethod =  String.class.getMethod("length");
> lengthMethod.invoke("abcdefg")
7

// "abcdefg".substring(2, 5) => "cde"
> Method substr = String.class.getMethod("substring",
                    Integer.TYPE, Integer.TYPE);
> substr.invoke("abcdefg", 2, 5)
"cde"
```

Reflection example 2

- Calling `translate` on a `Point` object:

```
// get the Point class object; create two new Point()s
Class<Point> clazz = Point.class;
Point p  = clazz.newInstance();
Point p2 = clazz.newInstance();

// get the method Point.translate(int, int)
Method trans = clazz.getMethod("translate",
        Integer.TYPE, Integer.TYPE);

// call p.translate(4, -7);
trans.invoke(p, 4, -7);

// call p.getX()
Method getX = clazz.getMethod("getX");
double x = (Double) getX.invoke(p);       // 4.0
```

Modifier static methods

```
if (Modifier.isPublic(clazz.getModifiers()) { ...
```

static method	description
`isAbstract(`**mod**`)`	is it declared abstract?
`isFinal(`**mod**`)`	is it declared final?
`isInterface(`**mod**`)`	is this type an interface?
`isPrivate(`**mod**`)`	is it private?
`isProtected(`**mod**`)`	is it protected?
`isPublic(`**mod**`)`	is it public?
`isStatic(`**mod**`)`	is it static?
`isSynchronized(`**mod**`)`	does it use the synchronized keyword?
`isTransient(`**mod**`)`	is the field transient?
`isVolatile(`**mod**`)`	is the field volatile?
`toString(`**mod**`)`	string representation of the modifiers such as "public static transient"

Field class methods

method	description
get(**obj**)	value of this field within the given object
getBoolean(**obj**), getByte(**obj**) getChar(**obj**), getDouble(**obj**) getFloat(**obj**), getInt(**obj**) getLong(**obj**), getShort(**obj**)	versions of get that return more specific types of data
getDeclaringClass()	the class that declares this field
getModifiers()	whether the field is private, static, etc.
getName()	field's name as a string
getType()	a Class representing this field's type
set(**obj**, **value**)	sets the given object's value for this field
setBoolean(**obj**, **value**), setByte(**obj**, **value**), ...	versions of set that use more specific types of data
toString()	string matching the field's declaration

Constructor methods

method	description
`getDeclaringClass()`	the class that declares this constructor
`getExceptionTypes()`	any exceptions the constructor may throw
`getModifiers()`	whether the constructor is public, etc.
`getName()`	constructor's name (same as class name)
`getParameterTypes()`	info about the constructor's parameters
`getReturnType()`	info about the method's return type
`newInstance``(`**params**`)`	calls this constructor, passing the given parameter values; returns object created
`toString()`	string matching the constructor's header

Array class methods

static method	description
get(**array**, **index**)	value of element at given index of array
getBoolean(**array**, **index**), getChar(**array**, **index**), getDouble(**array**, **index**), getInt(**array**, **index**), getLong(**array**, **index**), ...	versions of get that return more specific types of data
getLength(**array**)	length of given array object
newInstance(**type**, **length**)	construct new array with given attributes
set(**array**, **index**, **value**)	sets value at given index of given array
setBoolean(**array**,**index**,**value**), setChar(**array**,**index**,**value**),...	versions of set that use more specific types of data

- The Class object for array types has a useful method:

static method	description
getComponentType()	a Class object for the type of elements

Invocation exceptions

- If something goes wrong during reflection, you get exceptions.
  - Almost all reflection calls must be wrapped in try/catch or throw.
  - Example: `ClassNotFoundException`, `NoSuchMethodError`
- When you access a private field, you get an `IllegalAccessException`.
  - Else reflection would break encapsulation.
- When you call a method via reflection and it crashes, you will receive an `InvocationTargetException`.
  - Inside this is a *nested exception* containing the actual exception thrown by the crashing code.
  - You can examine the nested exception by calling `getCause()` on the invocation target exception.

Misuse of reflection

- Some programmers who learn reflection become overly enamored with it and use it in places where it wasn't intended.
  - Example: Passing a `Method` as a way to get a "function pointer."
  - Example: Checking the `Class` of values as a way of testing types.
  - Reflection code is usually bulky, ugly, brittle, and hard to maintain.
- Reflection is for certain specific situations only.
  - When you don't know what type to use until runtime.
  - When you want to be able to dynamically create or load classes while a program is running (example: CSE 14x Practice-It tool).
  - When you want to check information about a particular type.
  - When you want to write testing/grading libraries like JUnit.

Reflection examples

- The CSE 142 Critters simulator uses reflection to load all of the student's critter animal classes into the system.
  - Uses reflection to look for all classes with a superclass of `Critter`, constructs new instances of them, and adds them to the simulator.

- The CSE 14x Practice-It! tool uses reflection to dynamically compile, load, run, and test program code submitted by students.
  - The student's code is injected into a randomly named new class.
  - The class is written to disk, compiled, and loaded into the VM.
  - By reflection, the methods/code in the class are executed and tested.
  - Test results are gathered and shown to the student.

Reflection exercise

- Write a JUnit test to help grade the internal correctness of a student's submitted program for a hypothetical assignment.
  - Make the tests fail if the class under test has any of the following:
    - more than 4 fields
    - any non-`private` fields
    - any fields of type `ArrayList`
    - fewer than two `private` helper methods
    - any method that has a `throws` clause
    - any method that returns an `int`
    - missing a zero-argument constructor