Introduction to Programming
Part I

Lecture 14
Java Miscellaneous

Eugene Zouev
Fall Semester 2020
Innopolis University

What We Have Learnt

- Classes and class instances
- Value types and reference types
- Encapsulation, overloading
- Inheritance: single & multiple
- Static & dynamic types
- Method overriding
- Polymorphism
- Casts & type checks
- Abstract classes & methods
- Packages
- Exceptions
- Interfaces

Plan for the rest of the course

- 12 Java generics
- 13 Java lambdas
- 14 Java miscellaneous

The Plan for Today

- Enumeration Types
- Assertions
- Serialization
- Reflection
- Annotations

Enumerations

From the "C" part of the course

An example:
Suppose we are going to control the traffic lights
with three states: **red**, **yellow** and **green**.
How do we do that?

Enumerations

From the "C" part of the course

An example:
Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
```

This variable serves as a model of a traffic lights

Enumerations

From the "C" part of the course

An example:
Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
tl = 777;
```

Why these numbers?
Why not 4, 12, 78?

This variable serves as a model of a traffic lights

What happens if we write this ?

Enumerations

From the "C" part of the course

An example:
Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
tl = 777;
```

Why these numbers?
Why not 4, 12, 78?

This variable serves as a model of a traffic lights

What happens if we write this ?

Advanced solution

This is **enumeration type**!

```
enum Lights {
    green,
    yellow,
    red
};
...
Lights tl;
...
tl = 777; // ERROR
```

These are **enumerators**

Enumerations

From the "C" part of the course

An example:
Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
tl = 777;
```

Why these numbers?
Why not 4, 12, 78?

This variable serves as a model of a traffic lights

What happens if we write this ?

Advanced solution

This is **enumeration type**!

```
enum Lights {
    green,
    yellow,
    red
};
...
Lights tl;
...
tl = 777; // ERROR
```

These are **enumerators**

In general, we are not interested in actual values behind `green`, `yellow` & `red`!

However…
"Behind the scenes", the enumerator values are just integers, starting from 0.

Enumerations: More Examples

A model of a compass

```
enum Compass
{
   NORTH,
   SOUTH,
   EAST,
   WEST
}
```

Week days

```
public enum Day
{
   SUNDAY,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY
}
```

Enumerations: More Examples

A model of a compass

```
enum Compass
{
   NORTH,
   SOUTH,
   EAST,
   WEST
}
```

Enumeration members – **enumerators** – are actually *constants*.
Historically (from C or even from the Assembler era) constant names were written with UPPERCASE letters.

This is NOT a requirement – just a tradition…

Week days

```
public enum Day
{
   SUNDAY,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY
}
```

Enumerations: More Examples

```
public enum Day
{
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
}
```

```
public void tell_it_like_it_is(Day day)  // ☺
{
  switch (day) {
    case MONDAY:
      System.out.println("Mondays are bad.");
      break;
    case FRIDAY:
      System.out.println("Fridays are better.");
      break;
 // ...
    default:
       System.out.println("Midweek days are so-so.");
       break;
  }
}
```

Enumerations

Java extensions

Java enum types are much more powerful than their counterparts in other languages.

- Enum members can be initialized.
- The enum class body can include methods and other fields.
- The compiler automatically adds some special methods when it creates an enum.
- Enum members can have… bodies (!)

Enumerations

Java extensions

Enum members can be initialized

```
enum Coin
{
    PENNY    , NICKEL    , DIME     , QUARTER     ;


}
```

Enumerations

Java extensions

Enum members can be initialized

The value associated with the enumerator

Values (in cents) of American coins

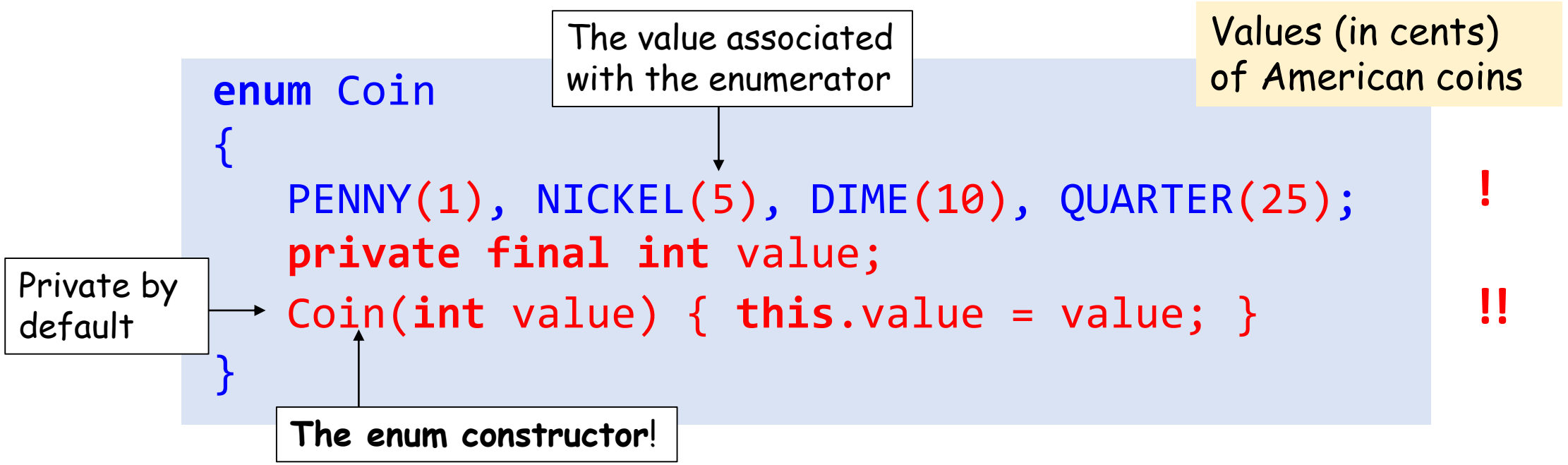
```
enum Coin
{
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);


}
```

!

Enumerations

Java extensions

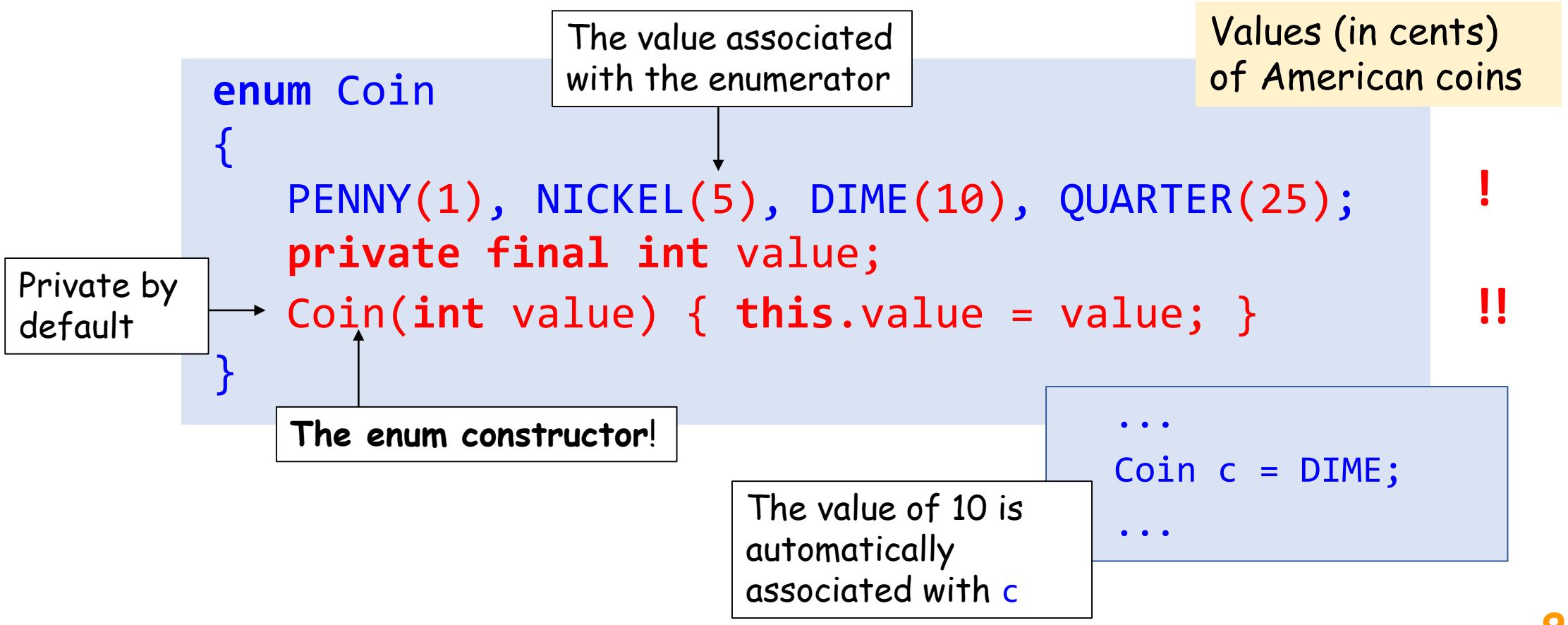
Enum members can be initialized



Enumerations

Java extensions

Enum members can be initialized



Enumerations

Java extensions

Enumerations can have methods

```
enum Coin
{
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    Coin(int value) { this.value = value; }

    private final int value;
    public int value() { return value; }
}
```

```
...
Coin c = DIME;
int v = c.value();
...
```

Returns 10

Enumerations

Java extensions

Enum predefined methods

From the Java Reference Manual

```
public static E[] values();
```

Returns an array containing the constants of this enum type, in the order they're declared. This method may be used to iterate over the constants:

Enumerations

Java extensions

Enum predefined methods

From the Java Reference Manual

```
public static E[] values();
```

Returns an array containing the constants of this enum type, in the order they're declared. This method may be used to iterate over the constants:

```
public class Test {
  enum Season { WINTER, SPRING, SUMMER, FALL }

  public static void main(String[] args) {
    for (Season s : Season.values())
      System.out.println(s);
  }
}
```

Output

```
WINTER
SPRING
SUMMER
FALL
```

Enumerations

Java extensions

Enum members can have bodies

```
enum Operation {
  PLUS   { double eval(double x, double y) { return x + y; } },
  MINUS  { double eval(double x, double y) { return x - y; } },
  TIMES  { double eval(double x, double y) { return x * y; } },
  DIVIDE { double eval(double x, double y) { return x / y; } };

  // Each constant supports an arithmetic operation
  abstract double eval(double x, double y);

  public static void main(String args[]) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
  }
}
```

Enumerations

Java extensions

Enum members can have bodies

```
enum Operation {
  PLUS   { double eval(double x, double y) { return x + y; } },
  MINUS  { double eval(double x, double y) { return x - y; } },
  TIMES  { double eval(double x, double y) { return x * y; } },
  DIVIDE { double eval(double x, double y) { return x / y; } };

  // Each constant supports an arithmetic operation
  abstract double eval(double x, double y);

  public static void main(String args[]) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
  }
}
```

Output

```
2.0 PLUS 4.0 = 6.0
2.0 MINUS 4.0 = -2.0
2.0 TIMES 4.0 = 8.0
2.0 DIVIDE 4.0 = 0.5
```

Assertions

- An **assertion** is a boolean expression that a programmer confirms is true at some point during the execution of a program.
- If the expression is false, the program throws `AssertionError`, which typically terminates the program and reports an error message.
- Assertions are widely used by programmers to detect bugs and gain confidence in the correctness of programs.
- They also serve to document the programmer's intent.
- They are not usually used for released code.

Assertion: the simple case

```
assert Condition ;
```

Condition is an expression that evaluates to a boolean value

Assertion: the simple case

```
assert Condition ;
```

Condition is an expression that evaluates to a boolean value

Example: you might call a method that should always return a positive integer value.

```
...
int res = someObj.someMethod();
// How to ensure that res is always positive?

assert res>0;
...
```

- At run time, if the condition is **true**, no other action takes place.
- However, if the condition is **false**, then AssertionError is thrown.

Assertion: the extended case

```
assert Condition : Expression ;
```

- *Condition* is an expression that evaluates to a boolean value.
- *Expression* is any expression whose value gets passed as an argument to the `AssertionError` constructor.

Assertion: the extended case

```
assert Condition : Expression ;
```

- *Condition* is an expression that evaluates to a boolean value.
- *Expression* is any expression whose value gets passed as an argument to the AssertionError constructor.

```
...
int res = someObj.someMethod();
// How to ensure that res is always positive?

assert res>0 : "n is not positive!";
...
```

In case `res <=0` the value of the expression is converted to its string representation, if needed (`toString()` is invoked for that) and displayed in the message

```
Exception in thread "main" java.lang.AssertionError: n is not
positive!
        at AssertDemo.main(AssertDemo.java:17)
```

Assertions: Details

- By default, assertions are disabled.
- To enable assertion checking at run time, you must specify the `-enableassertions` option (**`-ea`** for short).

  For example, to enable assertions for some program `Program`, launch the program like this:

```
java -ea Program
```

Assertions: Comments

- An important point to understand about assertions is that you must not rely on them to perform any action actually required by the program.

  The reason is that normally, released code will be run with assertions disabled.

- Assertions can be quite useful because they streamline the type of error checking that is common during development.
- With assert, you don't have to remove the assert statements from your released code.

Serialization

Sometimes it's necessary to save the current state of a running program in a persistent storage area (e.g., on a disk file).

Serialization

Sometimes it's necessary to save the current state of a running program in a persistent storage area (e.g., on a disk file).

The reasons:

- We would like to stop long-term execution and the continue running after some time…
- We would like to keep some intermediate results of execution…

…So, that at a later time we could restore the saved state of a program (its objects) – via **deserialization** - and continue execution.

Serialization

One more reason:

- Serialization is also needed to implement **Remote Method Invocation** (RMI).
- RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
- An object may be supplied as an argument to that remote method.
- The sending machine serializes the object and transmits it.
- The receiving machine deserializes it.

Serialization: Example

Part 1:
Objects to be serialized

```
class myClass implements Serializable
{
   String s;
   int i;
   double d;

   public myClass(String s, int i, double d) {
     this.s = s;
     this.i = i;
     this.d = d;
   }

   String toString() {
     return "s=" + s + "; i=" + i + "; d=" + d;
   }
}
```

Classes whose objects are to be serialized must implement the `Serializable` interface

The extra constant member is automatically added to the class with this interface implemented:
`static final long serialVersionUID = …`

The method `toString()` is not mandatory; just for experimenting

Serialization: Example

Part 2:
Serialization scheme

Simplified example: without catching possible exceptions

```
import java.io.*;

class Program {
  public static void main(String args[]) {

    var out = new ObjectOutputStream(new FileOutputStream("storage"));
    var myObj = new myClass("test",777,2.7e10);
    System.out.println("myObj: " + myObj);

    out.writeObject(myObj);

    ...
  }
}
```

Creating a disk file for storing objects

The method `writeObject()` performs serialization of `myObj` to the storage "`storage`"

Serialization: Example

Part 3:
Deserialization scheme

Simplified example: without catching possible exceptions

```
import java.io.*;

class Program {
  public static void main(String args[]) {

    ...
    var receive = new ObjectInputStream(new FileInputStream("storage"));
    var myObjReceived = (myClass)receive.readObject();
    System.out.println("myObjReceived: " + myObjReceived);
  }
}
```

Opening the disk file created for serializing before: now for reading

Notice that `readObject()` method knows nothing about the type of the object being deserialized; therefore, we have to cast it to its real type.

Serialization: Details

- Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a **directed graph**.
- There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves.
- The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all other referenced objects are recursively located and serialized.
- Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

Reflection
or Introspection

Reflection (in programming) is the ability to dynamically work with all program components:

- To dynamically get information about classes, their members and methods
  "The ability or software to analyze itself"
- To dynamically update class structure adding or removing members and methods
- To create classes with members and methods "on the fly", dynamically, while executing a program
- Create instances of dynamically created classes and invoke such methods!

Reflection

- Usually, the reflection feature is not a part of a language but is supported by a (part of a) language standard library.
- ***C#***: `System.Reflection`, `System.Reflection.Emit`
- ***C++***: WIP ("work in progress" ☺)
- **Java**: The Java Core Reflection API,
  `package` `java.lang.reflect`

Reflection: a Simple Example

```
import java.lang.reflect.*;
class ReflectDemo
{
  public static void main(String args[])
  {
    Class<?> c =
          Class.forName("java.awt.Dimension");









  }
}
```

By using reflection, one can determine what methods, constructors, and fields a class supports.

Reflection: a Simple Example

```
import java.lang.reflect.*;
class ReflectDemo
{
  public static void main(String args[])
  {
    Class<?> c =
          Class.forName("java.awt.Dimension");

    Constructor<?> ctors[] = c.getConstructors();

    Field fields[] = c.getFields();

    Method methods[] = c.getMethods();
  }
}
```

By using reflection, one can determine what methods, constructors, and fields a class supports.

Reflection: a Simple Example

```
import java.lang.reflect.*;
class ReflectDemo
{
  public static void main(String args[])
  {
    Class<?> c =
          Class.forName("java.awt.Dimension");

    Constructor<?> ctors[] = c.getConstructors();

    Field fields[] = c.getFields();

    Method methods[] = c.getMethods();
  }
}
```

By using reflection, one can determine what methods, constructors, and fields a class supports.

```
System.out.println("Constructors:");
for (int i=0; i<ctors.length(); i++)
  System.out.println(ctors[i]);
```

```
System.out.println("Fields:");
for (int i=0; i<fields.length(); i++)
  System.out.println(fields[i]);
```

```
System.out.println("Methods:");
for (int i=0; i<methods.length(); i++)
  System.out.println(methods[i]);
```

Reflection: a Simple Example

The output

```
Constructors:
 public java.awt.Dimension(int,int)
 public java.awt.Dimension()
 public java.awt.Dimension(java.awt.Dimension)
Fields:
 public int java.awt.Dimension.width
 public int java.awt.Dimension.height
Methods:
 public int java.awt.Dimension.hashCode()
 public boolean java.awt.Dimension.equals(java.lang.Object)
 public java.lang.String java.awt.Dimension.toString()
 public java.awt.Dimension java.awt.Dimension.getSize()
 public void java.awt.Dimension.setSize(double,double)
 public void java.awt.Dimension.setSize(java.awt.Dimension)
 public void java.awt.Dimension.setSize(int,int)
 public double java.awt.Dimension.getHeight()
```

Annotations

- Annotations are the means for adding some extra information to classes, methods and class members.

  Such information is often called **metadata**.

  Introduced in Java 5

  JSR #175: http://www.jcp.org/en/jsr/detail?id=175
- Look like a comment but better than just a comment.
- Some of annotations are predefined by the JDK:

```
@Deprecated
@SuppressWarnings
@Override
...
```

- It's possible to specify own annotations.

Predefined Annotations

`@Deprecated` is used to tell the developer that a function shouldn't be used anymore. It generally means that in a later version of the code, that function will go away.
When a developer still uses this function, the compiler will warn the developer about this.

Predefined Annotations

@Deprecated is used to tell the developer that a function shouldn't be used anymore. It generally means that in a later version of the code, that function will go away.
When a developer still uses this function, the compiler will warn the developer about this.

```
@Deprecated
double myOldSin(double x)
{
  ...
}
double myNewBetterSin(double x)
{
  ...
}
```

```
...
double s = myOldSin(x);
...
```

Warning message!

```
...
double s = myNewBetterSin(x);
...
```

OK

Predefined Annotations

`@SuppressWarnings` tells the compiler that even though it might be inclined to warn the developer about what (s)he is doing in the next line, it's really is ok and the developer knows about the issue.

```
Object a = new Foo();


Foo b = (Foo)a;


@SuppressWarnings
Foo c = (Foo)a;
```

Predefined Annotations

`@SuppressWarnings` tells the compiler that even though it might be inclined to warn the developer about what (s)he is doing in the next line, it's really is ok and the developer knows about the issue.

```
Object a = new Foo();

Foo b = (Foo)a;

@SuppressWarnings
Foo c = (Foo)a;
```

Type cast: this is a potentially unsafe action.
The compiler issues a warning.

Type cast again.
The compiler keeps silence because a developer knows that it's OK.

Predefined Annotations

@Override tells the compiler that a developer intends to have a function override the parent's version of it.

If there is a typo, the compiler knows there's a problem and will throw an error.

@Override also tells the developer that this code is overriding a parent's version.

```
public class Foo extends Bar {
   public String tooString() {
      ...
   }

   @Override
   public String tuString() {
      ...
   }
}
```

Predefined Annotations

`@Override` tells the compiler that a developer intends to have a function override the parent's version of it.

If there is a typo, the compiler knows there's a problem and will throw an error.

`@Override` also tells the developer that this code is overriding a parent's version.

```
public class Foo extends Bar {
   public String tooString() {
      ...
   }

   @Override
   public String tuString() {
      ...
   }
}
```

This is the typo but the compiler doesn't understand this; it assumes that this is a new "non-overriding" method

This is also the typo but compiler does understand this: it detects that there is no `tuString` method in the superclass

User-defined Annotations

The most annotations are only compile-time entities:
they are typically used by third-party tools that work
with Java code:

- Configuration tools like `ant`
- Static analyzers
- Pretty-printers
- ...

User-defined Annotations

The most annotations are only compile-time entities: they are typically used by third-party tools that work with Java code:

- Configuration tools like ant
- Static analyzers
- Pretty-printers
- ...

The simplest example of annotation declaration

```
public @interface MyAnnotation { }
```

User-defined Annotations

Example: we define three annotations that indicate the current state of our code that is "under construction"

```
public @interface Preliminary { }
public @interface RawCode { }
public @interface Ready { }
```

User-defined Annotations

Example: we define three annotations that indicate the current state of our code that is "under construction"

```
public @interface Preliminary { }
public @interface RawCode { }
public @interface Ready { }
```

…And a configuration tool can analyze class annotations (**via reflection!**) and select classes that are, say, ready for being included into release…

Having such annotations we can "mark" our classes correspondingly…

```
@Preliminary
public class Class1 { ... }

@RawCode
public class Class2 { ... }

@Ready
public class Class3 { ... }
```

That's It With Miscellaneous …

That's It With Miscellaneous …

…That's It With the Talk…

That's It With Miscellaneous …

…That's It With the Talk…

…And That's It With the Course!


WELL,
THAT'S ALL FOLKS!
THANKS FOR WATCHING
WE HOPE YOU LEARNED SOMETHING NEW
...and listening