Java Basics II

COMP3111/H Tutorial2

Java Object and Class

- Object-oriented programming (OOP)
- Exception handling

OO Programming Concepts

- Object-oriented programming (OOP) involves programming using **objects**
- An *object* represents an entity in the real world that can be distinctly identified
  - E.g., a student, a desk, a circle, a button

Fields and Methods

- An *object* has a unique identity, state, and behaviors
  - The *state* of an object consists of a set of *data* **fields** (also known as *properties*) with their current values.
  - The *behavior* of an object is defined by a set of **methods**.

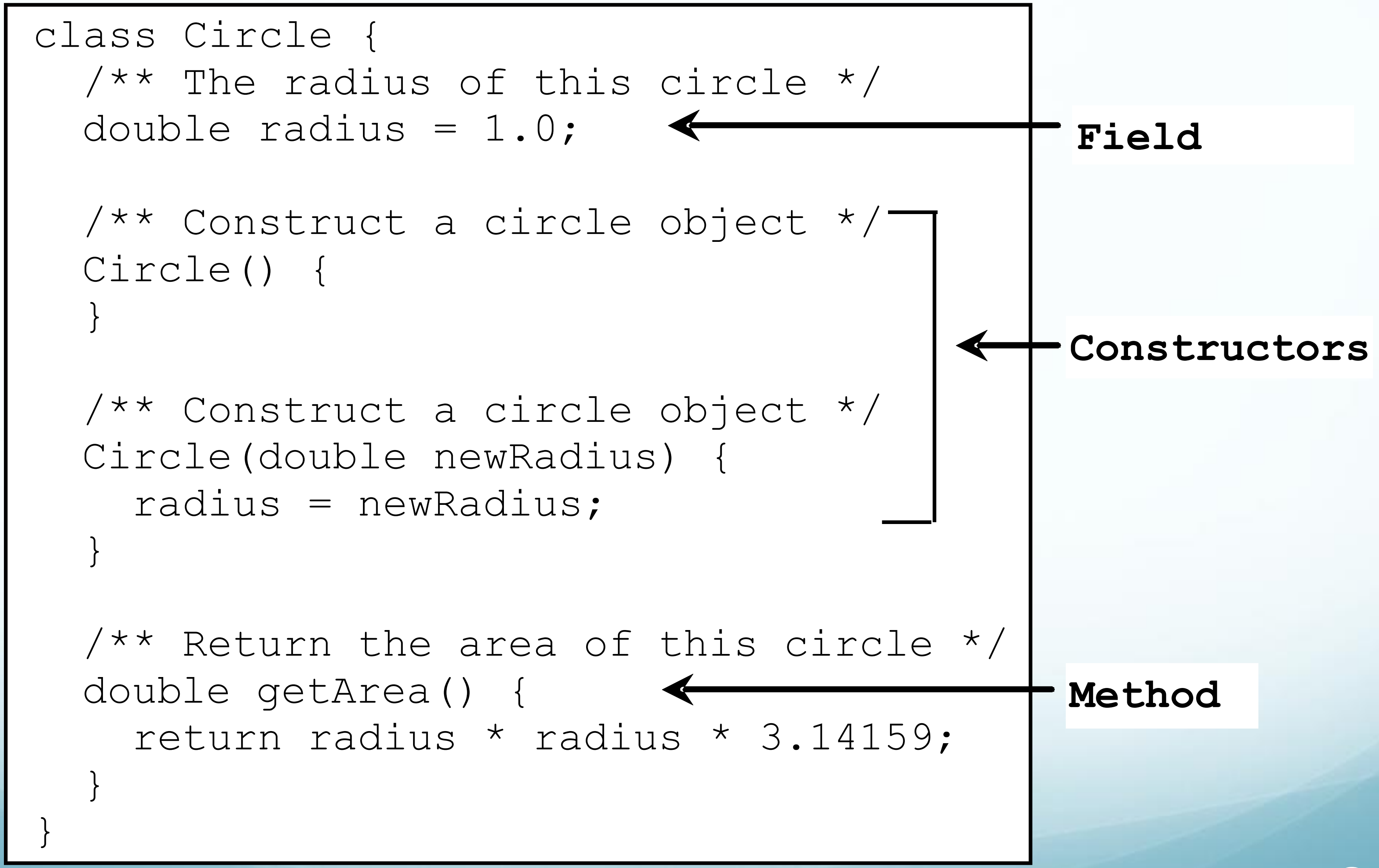
Classes

Classes are constructs that define objects of the same type.

A Java class uses **variables** to define data **fields** and **methods** to define **behaviors**

Additionally, a class provides a special type of methods, known as **constructors**, which are invoked to construct objects from the class.

Classes



```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Constructors

- Constructors are a special kind of methods that are invoked to construct objects.

```
Circle() {
}
Circle(double newRadius) {
  radius = newRadius;
}
```

Constructors (cont.)

- Constructors play the role of **initializing** objects
- Constructors must have **the same name** as the class itself
- Constructors **do not** have a return type - not even void
- Constructors are invoked using the **new** operator when an object is created
- A constructor with no parameters is referred to as a ***no-arg constructor***

Examples

```
Circle c1 = new Circle();

Circle c2 = new Circle(5.0);
```

There is no “delete” operator

Java Virtual Machine uses garbage collector to free up memory automatically

Default Constructor

A class may be declared without constructors

In this case, a **no-arg** constructor with an empty body is implicitly declared in the class

This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly declared in the class*

Accessing fields and methods

- Referencing the object's data:

```
objectRefVar.data
```

e.g., `myCircle.radius = 5;`

(It works if the field is public – not recommended!)

- Invoking the object's method:

```
objectRefVar.methodName(arguments)
```

e.g., `int area = myCircle.getArea();`

Modifiers

- Modifiers allow programmers to control the behavior of classes, object constructors, methods, variables, and even blocks of code.
- **public**
  - Full access everywhere
- **static**

class	It can be called through *class name* directly without through *an instance of the object*
method or **field**	Method becomes a 'class' method invoked through the class name rather than an instance of the object. Even if new instances of the defining object are created, only one instance of the method or field exists

- A full list of java modifiers:
  - http://www.javacamp.org/javaI/modifier.html

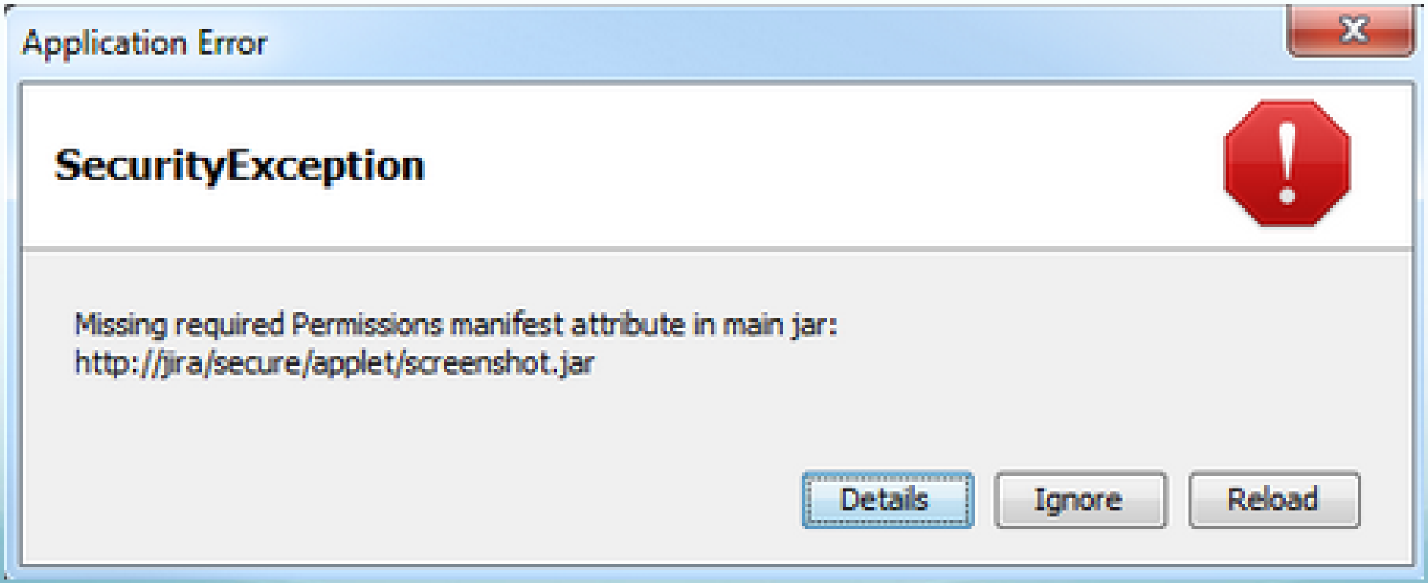
Example: The main method

```
class Hello {
public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

- The program runs start from the main method, and when the main method returns, the program exits.

Exception

- When a program has a runtime error, it terminates abnormally
- How can we handle the runtime error so that the program can continue to run or terminate gracefully?


Application Error

SecurityException

Missing required Permissions manifest attribute in main jar:
http://jira/secure/applet/screenshot.jar

Exception types

- **System errors** are thrown by JVM and represented in the Error class.
  - The Error class describes internal system errors. Such errors rarely occur.
  - If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exception and RuntimeException

- **Exception** describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.
- **RuntimeException** is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Syntax of exception

- In Java, the exception handling is built in to the language
- There is a keyword to **throw** an exception
- There is a keyword to **catch** an exception
- There is a keyword (**throws**) to declare that a method could throw an exception that the caller has to deal with

Declaring Exception

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
   throws IOException

public void myMethod()
   throws IOException, OtherException
```

Throwing Exception

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();

TheException ex = new TheException();
    throw ex;
```

Example

```
  /** Set a new radius */
  public void setRadius(double newRadius)  throws
    IllegalArgumentException
{
        if (newRadius >= 0)
                radius =  newRadius;
        else
                throw new IllegalArgumentException
                        ( "Radius cannot be negative");
 }
```

Catching Exception

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
Finally
{…}
```

Declaring, throwing and catching exceptions

```
public void methodA() throws SomeException, AnotherException
{
        //methodbody
}
public void methodB() throws CustomException
{
        //Methodbody
}
public void methodC()
{
        methodB();
        methodA();
}
```

Wrong example, methodC is invalid. Because methodA and methodB pass (or throw) exceptions, methodC must be prepared to handle them.

Declaring, throwing and catching exceptions

```
public void methodC() throws AnotherException,
SomeException
{
          try{
                   methodB();
         }
         catch (CustomException e) {
                   //handle it
         }
         methodA();
}
```

Revised version, the CustomException from methodB will be handled locally, while AnotherException and SomeException will be thrown to the caller to handle it.