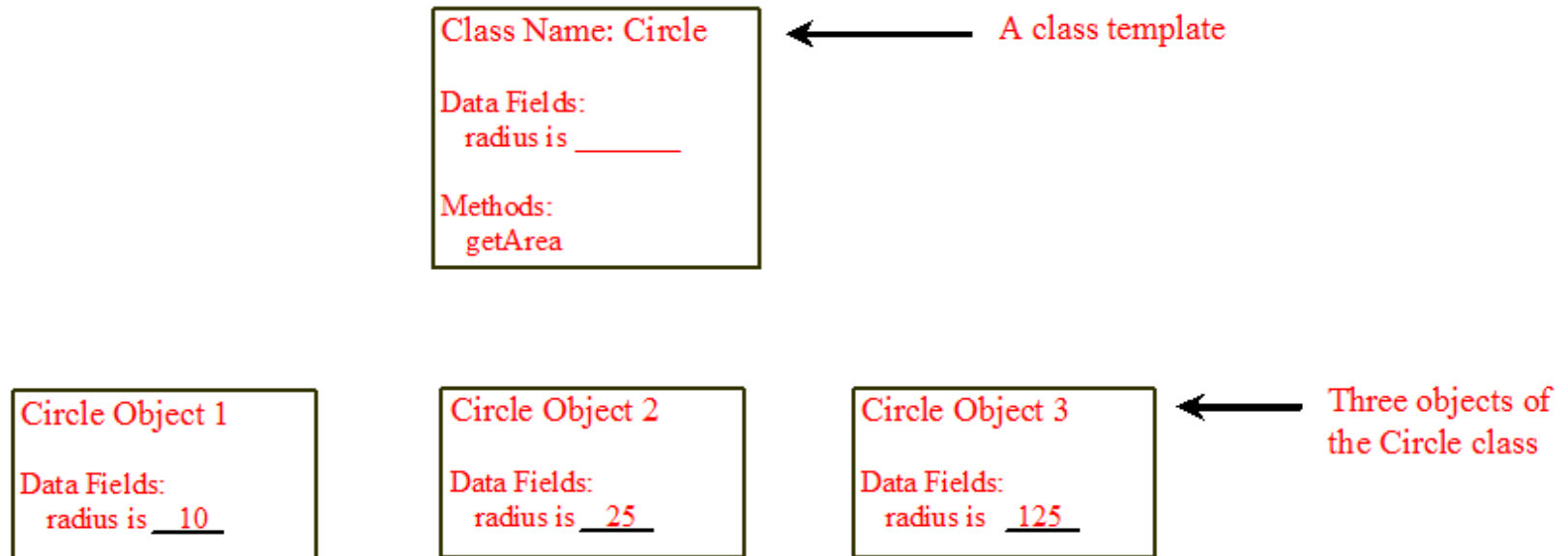


Objects and Classes

Classes and Objects



An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

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 method(s), known as constructors, which are invoked to construct objects from the class.

Classes

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

← Data field

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

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

← Constructors

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

← Method



Know Your Data item

```
class Car {  
    String licensePlate = ""; // fields  
    double speed = 0.0; // fields  
    static double maxSpeed; = 123.45; // fields  
    boolean isSpeeding(int i) { // parameters  
        double excess; // local variable  
        excess = maxSpeed - this.speed;  
        if (excess < 0) return true;  
        else return false;  
    }  
}
```

Fields define the state; local variables don't

Constructors

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

```
Circle() {  
}
```

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

Constructors

- 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. Constructors play the role of initializing objects.

Declaring Object Reference Variables

```
ClassName objectName;
```

Example:

```
Circle myCircle;
```


Accessing Objects

- Referencing the object's data:

`objectName.data`

e.g., `myCircle.radius`

- Invoking the object's method:

`objectName.methodName (arguments)`

e.g., `myCircle.getArea()`

Using **THIS**

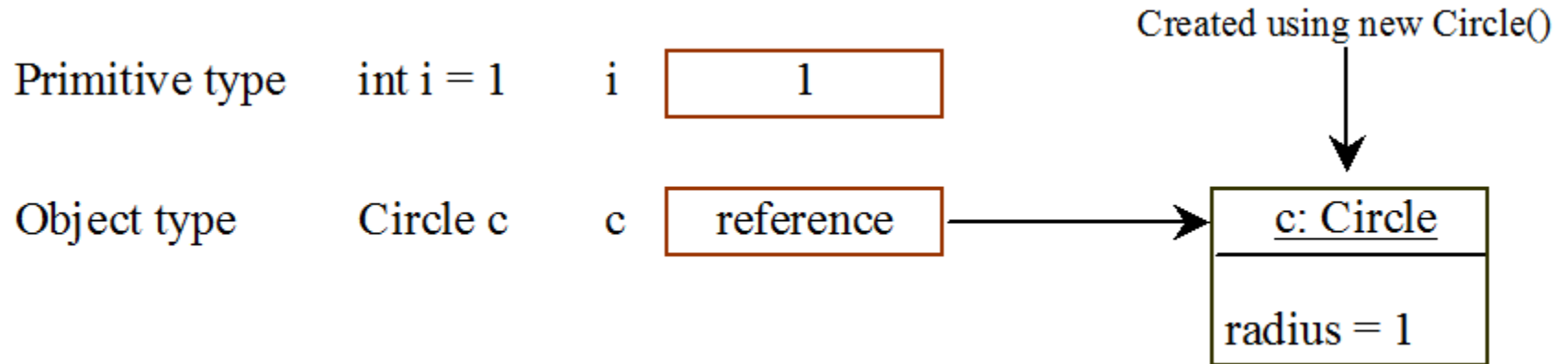
- **this** refers to the current object.

```
public Employee(String name, double salary, int year, int
                  month, int day) {
    this.name = name;
    this.salary = salary;
    ...
}
```

The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.

Differences between Variables of Primitive Data Types and Object Types



Copying Variables of Primitive Data Types and Object Types

Primitive type assignment $i = j$

Before:

i 1

j 2

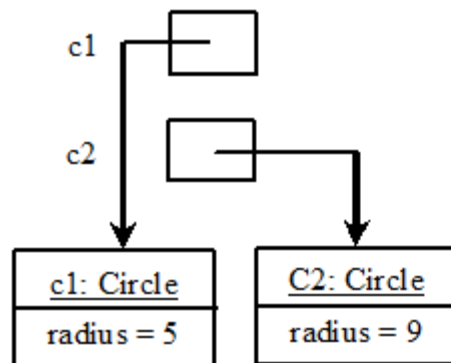
After:

i 2

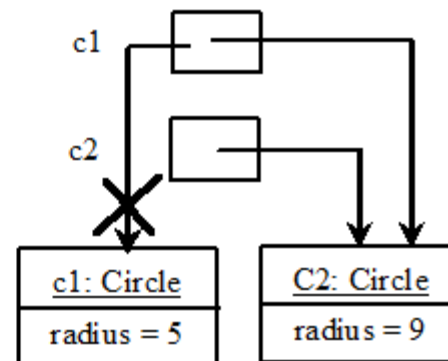
j 2

Object type assignment $c1 = c2$

Before:



After:



Garbage Collection

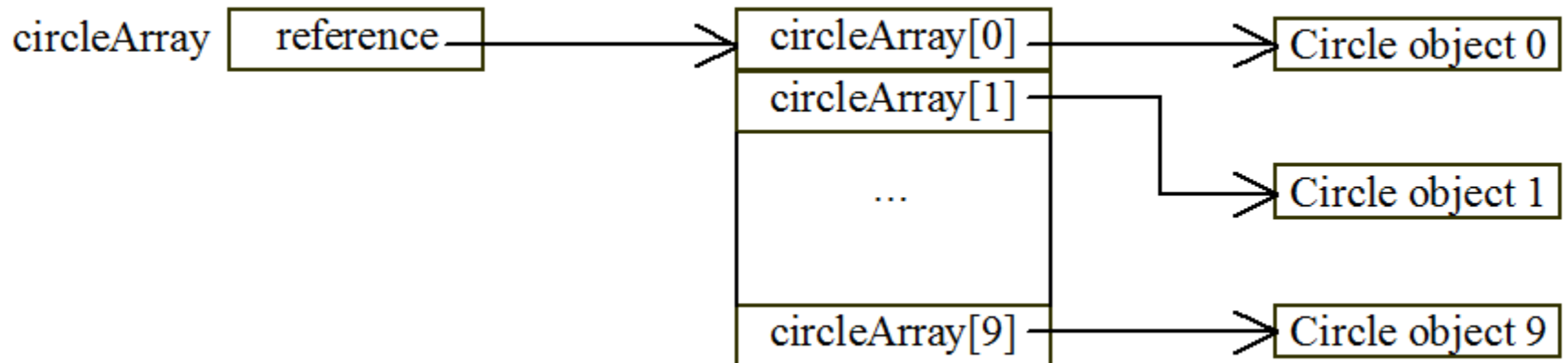
As shown in the previous figure, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`.

The object previously referenced by `c1` is no longer referenced.

This object is known as garbage. Garbage is automatically collected by JVM.

Array of Objects

```
Circle[] circleArray = new Circle[10];
```



Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

☞ `public`

The class, data, or method is visible to any class in any package.

☞ `private`

The data or methods can be accessed only by the declaring class.

The get and set methods are used to read and modify private properties.

Rule #1

Data Fields should always be private.

Data Fields should always be private.

Data Fields should always be private.

Data Fields should always be private.

Data Fields should always be private.

Hide Information Always!

Accessibility of Methods

- **public**: visible everywhere
 - **protected**: visible in package and in subclasses
 - Default (no modifier): visible in package
 - **private**: visible only inside class (more on this later)
-
- A method can access fields and methods that are visible to it.
 - **Public** method and fields of any class
 - **Protected** fields and methods of superclasses and classes in the same package
 - Fields and methods without modifiers of classes in the same packages
 - **Private** fields and methods of the same class.

package p1;

```
public class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void m3() {  
    }  
}
```

```
public class C2 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        can access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        can invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

package p2;

```
public class C3 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        cannot access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        cannot invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

package p1;

```
class C1 {  
    ...  
}
```

```
public class C2 {  
    can access C1  
}
```

package p2;

```
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

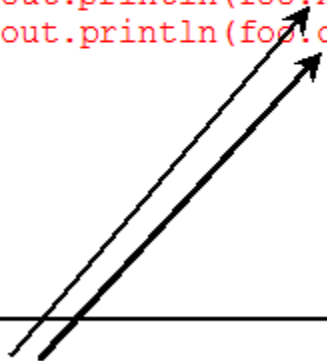
NOTE

An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class Foo {  
    private boolean x;  
  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
        System.out.println(foo.convert());  
    }  
  
    private int convert(boolean b) {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is OK because object foo is used inside the Foo class

```
public class Test {  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
        System.out.println(foo.convert(foo.x));  
    }  
}
```



(b) This is wrong because x and convert are private in Foo.

Types of Methods

- **Accessor** methods:

```
public String getName()  
{  
    return name;  
}
```

- **Mutator** methods:

```
Public void setSalary(double newSalary)  
{  
    salary = newSalary;  
}
```

Static Methods

- Declared with modifier **static**.
- It belongs to class rather than any individual object. Sometimes called class method.
- Usage: `className.staticMethod()` NOT `objectName.staticMethod()`

```
class Employee
{ public static int getNumOfEmployees()
  {
    return numOfEmployees;
  }
  private static int numOfEmployees = 0;
  ...
}
Employee.getNumOfEmployee(); // ok
Harry.getNumOfEmployee();   // not this one
```

Static Methods

- The main method

```
class EmployeeTest
{ public static void main(String[] args)
    { ...
    }
}
```

is always static because when it is called, there are not objects yet.

Command-Line arguments

Parameters of the **main** Method

```
public static void main(String args[])
{
    for (int i=0; i<args.length; i++)
        System.out.print(args[i]+" ");
        System.out.print("\n");
}
```

```
// note that the first element args[0]
// is not the name of the class, but the
// first argument
```


Function Overloading

- Can re-use names for functions with different parameter types

```
void sort (int[] array);  
void sort (double[] array);
```

- Can have different numbers of arguments

```
void indexof (char ch);  
void indexof (String s, int startPosition);
```

- Cannot overload solely on return type

```
void sort (int[] array);  
boolean sort (int[] array); // not ok
```

Resolution of Overloading

- Compiler finds best match
 - Prefers exact type match over all others
 - Finds “closest” approximation
 - Only considers **widening conversions**, not narrowing

- Process is called “resolution”

```
void b (int i, int j);
```

```
void b (double d, double e);
```

```
b(10, 8)    //will use (int, int)
```

```
b(3.5, 4)   //will use (double, double)
```

narrower

byte

short

char

int

long

float

double



broader

- **Widening conversion:** For a primitive data types, a value **narrower** data type can be converted to a value of a **broader** data type without loss of information.
- **Narrowing conversion:** Converting from a **broader** data type to a **narrower** data type is called narrowing conversion, which can result in loss of information.

Packages

- What are packages
- Creating packages
- Using packages

Packages

- A package consists of a collection of classes and interfaces
- Example:
 - Package java.lang consists of the following classes
 - Boolean Byte Character Class ClassLoader
Compiler
Double Float Integer Long Math Number Object
SecurityManager Short StackTraceElement
StrictMath String StringBuffer System Thread ...

Creating Packages

- To add to class to a package, say foo
 - Begin the class with the line
package foo;
 - This way we can add as many classes to foo as you wish
- Where should one keep the class files in the foo package?
 - In a directory name foo :
 - .../foo/{first.class, second.class, ...}
- Subpackages and subdirectories must match
 - All classes of foo.bar must be placed under .../foo/bar/
 - All classes of foo.bar.util must be under .../foo/bar/uti/

Creating Packages

- Classes that do not begin with “package ...” belongs to **the default package**:
 - The package located at the current directory, which has no name
 - Consider a class under .../foo/bar/
 - If it starts with “package foo.bar ”, it belongs to the package “foo.bar”
 - Else it belong to the default package

Using Packages

Use full name for a class: `packageName.className`

```
java.util.Date today = new java.util.Date();
```

Use `import` so as to use shorthand reference

```
import java.util.Date;  
Date today = new Date();
```

Can import all classes in a package with wildcard

- `import java.util.*;`
- Makes everything in the `java.util` package accessible by shorthand name: `Date`, etc.

Everything in `java.lang` already available by short name, no import necessary

Autoboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
- Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called *autoboxing*.

Integer number = 100; or

int m= 100; Integer number = m;

Autoboxing

- Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called **unboxing**.

```
Integer number = new Integer(100);  
int x = number;
```

Streams

- **System.in** is an InputStream which is typically connected to keyboard **input** of console programs.
- **System.out** is a PrintStream.

System.out normally **outputs** the data you write to it to the console.

- **System.err** is a PrintStream. System.err works like System.out except it is normally only used to output **error** texts.