PHP – En överlevnadsguide

Loopar och upprepning

Vill man upprepa kod flera gånger så istället för att skriva en massa rader så kan man lägga koden i ett kodblock som man sedan loopar. Det finns ett par olika typer av loopar, **FOR**-loop, **Foreach**-loop, **WHILE**-loop och **Do-While**-loop. Vilken typ som lämpar sig bäst beror på vad man vill göra. Ska man upprepa något ett bestämt antal gånger eller behöver en räknare så är **FOR**-loopen att föredra. Ska man upprepa något ett obestämt antal gånger så är **WHILE**-loopen att föredra. **Foreach**-loopen är kraftfull och mycket användbar när man jobbar med **objekt** (mer om objekt senare).

FOR-loopen

Består av tre delar. Först deklareras en s.k. **räknare** som ges ett startvärde. Sedan anges ett **villkor** som måste uppfyllas för att loopen ska köra. Sist så anges vad som ska ske efter varje upprepning (**iteration**).

Exempel:

```
for ($i=0; $i<10; $i++)
{
   echo $i;
}
```

Resultat: 0123456789

I exemplet ovan skrivs värdet på räknaren `$i` ut. Loopen börjar på 0 och körs så länge `$i` är mindre än 10. I slutet av varje upprepning (iteration) så ökas `$i` med ett. **OBS** variabler såsom räknaren som deklareras inuti en loop eller annan kodblock är bara definierad inom detta block och ”lever” så att säga bara där. Det går alldeles utmärkt att lägga andra loopar eller villkorssatser inuti en loop eller ändra startvärde, villkor och vad som ska ske efter varje upprepning.

Foreach-loop

Denna loop lämpar sig extra bra till att iterera genom varje element i ett **objekt** vilket är perfekt för **array**’er (fält) då dessa är objekt. Man kan se det som ”för varje element i arrayen kör följande block med kod”.

Exempel:

```
$djur=array("ko","katt","älg");
foreach ($djur as $d)
{
   echo $d . " ";
}
```

Resultat: ko katt älg

I exemplet ovan så för varje element i `$djur` så kopieras värdet till `$d` (tilldelningen är *by value*) och sedan körs kodblocket. **OBS** vill vi ändra på innehållet i en array med foreach så måste vi ändra tilldelningen till *by reference* med tecknet **&**. Se exemplet nedan.

Exempel:

```
$djur=array("ko","katt","älg");
foreach ($djur as &$d)
{
   $d = "En " . $d;
   echo $d . " ";
}
```

Resultat: En ko En katt En älg

OBS I exemplet ovan så ändras värdena i originalarrayen. Detta enbart p.g.a **&**-tecknet!

While-loop

Denna loop lämpar sig bra när man inte vet hur många gånger loppen ska upprepas. **OBS** Det är lätt att skapa oändliga loopar så var försiktig. While-loop består eg. bara av ett villkor och så länge detta uppfylls så upprepas loopen.

Exempel:

```
$tal = 0;
while ($tal < 10)
{
   echo $tal;
   $tal++;
}
```

Resultat: 0123456789

Exemplet ovan gör samma sak som FOR-loop-exemplet.

Det finns en alternativ syntax för while-loopen utan måsvingar som vissa anser är mer lättläst, se exempel nedan.

PHP – En överlevnadsguide

Exempel:

```
$tal = 0;
while ($tal < 10):
   echo $tal;
   $tal++;
endwhile;
```

Resultat: 0123456789

Samma exempel som innan fast med endwhile.

Do-While-loop

En vanlig While-loop kontrollerar villkoret före varje iteration. Ett logiskt alternativ är att göra det efter varje iteration. Alltså kommer koden i en Do-While-loop alltid att köras minst en gång medan koden i en While-loop kan

Exempel:

```
$tal = 0;
do {
   echo $tal;
}
while ($tal>1)
```

Resultat: 0

skippas helt ifall villkoret inte uppfylls.

Funktioner

En funktion består av kod som placeras i ett namngivet kodblock. Denna kod kan sedan när som helst *anropas* för att köras. Detta är perfekt för att återanvända kod, exempelvis ifall man ska utföra något ofta så istället för att upprepa flera rader kod så kan man lägga dessa i en funktion och anropa funktionen (en rad kod).

I PHP finns det massvis med inbyggda funktioner för att göra olika saker. Undersök alltid ifall det finns en färdig funktion i PHP innan du skriver en egen för att göra samma sak (i onödan).

Sträng-funktioner

Mycket i PHP handlar om stränghantering. Det finns massvis med användbara PHP-funktioner som underlättar vid stränghantering.

Exempel:

```
$length=srtlen("Kalle");
echo $length;
```

Resultat: 5

I Exemplet ovan används funktionen **strlen()** som *returnerar* antalet tecken i en textsträng. Textsträngen anges som *inparameter* inom parentesen.

Exempel:

```
$name = "Jesper";
$partial=substr($name,1,4);
echo $partial;
```

Resultat: espe

I Exemplet ovan används funktionen **substr()** som returnerar en del av en textsträng. Funktionen tar en textsträng som inparameter följt av startposition (index börjar på noll) och antal tecken som man vill ha av textsträngen.

Exempel:

```
$name = "JeSper";
echo strtoupper($name);
```

Resultat: JESPER

I Exemplet ovan används funktionen **strtoupper()** som tar en textsträng som inparameter och returnerar samma textsträng med versaler (stora bokstäver).

Exempel:

```
$name = "JeSper";
echo strtolower($name);
```

Resultat: jesper

I Exemplet ovan används funktionen **strtolower()** som tar en textsträng som inparameter och returnerar samma textsträng med gemener (små bokstäver).

PHP – En överlevnadsguide

Exempel:

```
$result = strpos("Jesper","e");
echo $result;
if(strpos("Hej", "f") ==false);
echo "Error";
```

Resultat

```
1
Error
```

I exemplet ovan används funktionen **`strpos()`** som används till att hitta första positionen av en s.k. *substring* (mindre del av en sträng eller tecken) i en annan sträng. I exemplet så letar vi efter första positionen för tecknet *e* i strängen *"Jesper"* vilket returnerar positionen 1 som är index i strängen. Finns inte den eftersökta textsträngen så returneras *false* vilket de två sista raderna illustrerar.

Matematik-funktioner

PHP innehåller en mängd användbara matematikfunktioner och användbara konstanter.

Exempel:

```
echo round(M_PI);
echo round(M_PI,3);
```

Resultat

```
3
3.142
```

I exemplet ovan används funktionen **`round()`** som används för att runda av decimaltal. I exemplet används matematikkonstanten **`M_PI`** som är ett fördefinierat namn för konstanten *pi*. Funktionen rundar av till heltal om inget annat anges men man kan även ange antal decimaler som ska användas (se rad 2).

Exempel:

```
print rand(1,10);
```

Resultat

```
4
```

I exemplet ovan används funktionen **`rand()`** som används för att generera ett slumptal, i detta fall ett tal mellan 1 och 10. Anges inga intervall så slumpas ett tal mellan 1 och 32767.

Array-funktioner

Array'er (eller *fält*) är kraftfulla datastrukturer som används väldigt mycket. Det finns flera användbara funktioner för att manipulera och hantera array'er.

Exempel:

```
$animals = array();
array_push($animals, "Cat");
array_push($animals, "Dog");
print count($animals);
```

Resultat

```
2
```

I exemplet ovan används funktionen **`array()`** som för att skapa en array. Sedan används funktionen **`array_push()`** som används för att lägga till ett element i slutet av en array. Sist så används funktionen **`count()`** returnerar antalet element i en array.

Exempel:

```
$tal = array(1,6,3,8,5,9,0);
sort($tal);
print join(",", $tal);
```

Resultat

```
0,1,3,5,6,8,9
```

I exemplet ovan används funktionen **`sort()`** som sorterar en array. En annan variant är funktionen **`rsort()`** (*reverse* sort) som sorterar i omvänd ordning. Funktionen **`join()`** används till att lägga ihop alla element i en array med valfritt tecken som *lim* (i detta fall komma-tecknet)

Skapa egna funktioner

Genom att skapa egna funktioner så kan vi återanvända kod och skapa snyggare kod samt skriva mindre.

Exempel:

```
function helloWorld() {
   echo "Hello world!";
}
helloWorld();
```

Resultat

```
Hello World!
```

I exemplet ovan skapar vi en egen funktion med namnet **`helloWorld`** som skriver ut en textsträng. Vi anropar funktionen genom att skriva **`helloWorld()`**. Observera att vår

PHP – En överlevnadsguide

funktion inte tar emot någon *inparameter (argument)* och returnerar inte heller något. Oftast vill man att en funktion ska returnera ett värde för att kunna använda funktionen på ett mer flexibelt sätt.

Exempel:

```
function test() {
   return "Hej!";
}
echo test();
```

Resultat: Hej!

I exemplet ovan skapar vi en egen funktion med namnet **`test`** som till skillnad från förra exemplet *returnerar* textsträngen *Hej!*. Detta sker med kodordet **`return`**. Det går alldeles utmärkt att returnera variabler och objekt också. När vi anropar funktionen så skrivs inget ut utan vi får en textsträng som vi skriver ut med echo.

Exempel:

```
function aboutMe($name, $age) {
  echo "Hej $name, du är $age";
}
aboutMe("Jesper", 35);
```

Resultat: Hej Jesper, du är 35 år

I exemplet ovan skapar vi en egen funktion med namnet **`aboutMe`** som tar två *inparametrar*. När vi anropar funktionen så anges inparametrarna i parentesen skiljt med kommatecken.

OBJEKT

PHP är ett objektorienterat språk och man pratar ibland om objektorienterad programmering (OOP). Ett *objekt* kan man se som ett paket som kan innehålla variabler och funktioner. När man pratar om objekt så kallar man variabler som hör till ett objekt för *egenskaper, medlemsvariabler* eller *attribut* och funktioner kallas ibland för *metoder*.
I de allra flesta objektorienterade programmeringsspråken så är faktiskt allt objekt, även vanliga "variabler" och funktioner. För att skapa egna objekt så måste vi skapa *klasser*. En *klass* kan man se som en byggritning/beskrivning av ett objekt. Skapar man nya objekt så kallas det för en *instans* av en klass.

Exempel:

```
class Frukt {
  public $antal = 3;
  public $typ;

}
$banan = new Frukt();
$banan->typ = "banan";
print $banan->antal;
print $banan->typ;
```

Resultat:
3
banan

I exemplet ovan skapas klassen *Frukt* med det reserverade ordet **`class`**. Klassen har två medlemsvariabler. Ett nytt objekt **`$banan`** av typen Frukt skapas med reserverade ordet **`new`**. För att komma åt medlemsvariabler används **`->`**.

Exempel:

```
class Frukt {
  public function
  __construct($t) {
      $this->typ=$t;
    }

  public $typ;

}
$banan = new Frukt("banan");
print $banan->typ;
```

Resultat: banan

Objekt kan som sagt även innehålla funktioner. I exemplet ovan skapas en speciell funktion med det reserverade namnet **`__construct`**. Denna funktion är en *konstruktor* som körs när man skapar ett nytt objekt. En konstruktor är användbar för att initialisera ett objekt och ge dess medlemsvariabler lämpliga värden. Det reserverade ordet **`$this`** hänvisar till det aktuella objektet. När ett nytt objekt med

PHP – En överlevnadsguide

namn **$banan** av typen **Frukt** skapas med det reserverade ordet **new** så anropas konstruktorn. Konstruktorn i exemplet tar en textsträng som inparameter och tilldelar medlemsvariabeln **$typ** värdet av inparametern.

Exempel:

```
class Frukt {
  public function skriv($t) {
      echo $t;
    }
}
$banan = new Frukt();
$banan -> skriv("Hej!");
```

Resultat

```
Hej!
```

Ett sista exempel för att illustrera funktioner i ett objekt (metoder). Här skapas ett nytt objekt **$banan** av typen **Frukt**. Sedan anropas metoden **skriv** som tar en inparameter och skriver ut den.

Objekt- och klassmetoder

Det finns ett antal nyttiga metoder som är användbara då vi jobbar med objekt.

Exempel:

```
class Person {
  function __construct($name) {
    $this->name = $name;
  }
  public function dance() {
    return "I'm dancing!";
  }
}
$me = new Person("Kalle");
if (is_a($me, "Person")) {
  echo "I'm a person, ";
}
if (property_exists($me,
"name")) {
  echo "I have a name, ";
}
if (method_exists($me,
"dance")) {
 echo "and I know how to
dance!";
```

Resultat

```
I'm a person, I have a name, and I know how to dance
```

Exemplet visar tre användbara metoder. **is_a()** kontrollerar ifall ett objekt är en instans av en viss klass. Metoden **property_exists()** kontrollerar ifall objektet har namngiven medlemsvariabel. Metoden **method_exists()** kontrollerar ifall objektet har namngiven metod. Alla metoder anropas med objektets namn först följt av en textsträng.

Arv

Inom OOP så kan man använda något som kallas *arv* (inheritance på engelska). Detta är användbart då vi kanske har en klass som är en specialvariant av en annan klass. Exempelvis ifall vi har en generell klass **Bil** och en klass **Lastbil**. Då är det praktiskt ifall vi kan ange att klassen **Lastbil** automatiskt ska ha många av egenskaperna och metoderna som **Bil** har. Detta kan vi göra genom arv och då används det reserverade ordet **extends** i PHP.

Exempel:

```
class Bil {
  public $harMotor = "Ja";

}
class Lastbil extends Bil {

}
$fordon = new Lastbil();
print $fordon->harMotor;
```

Resultat

```
Ja
```

I exemplet ovan har vi klassen **Lastbil** som ärver av klassen **Bil**. Klassen **Lastbil** ärver egenskaper och metoder från klassen **Bil**. I detta fall är **Lastbil** en *subklass (child)* till **Bil** som då kan kallas *superklass (parent eller basklass)*.

Ibland vill man skriva över metoder som ärvs av en superklass. Detta görs enkelt genom att göra en ny implementation av metoden i subklassen.

PHP – En överlevnadsguide

Exempel:

```
class Bil {
  public function tuta() {
    echo "tuut";
  }

}
class Lastbil extends Bil {
  public function tuta() {
    echo "TUUT!";
  }
}
$fordon = new Lastbil();
$fordon->tuta();
```

Resultat: TUUT!

I exemplet ovan har superklassens (Bil) metod **`tuta()`** skrivits över av subklassens metod med samma namn.

Ibland kan det vara så att man vill förhindra subklasser från att skriva över metoder. Detta går att åstadkomma med det reserverade ordet **`final`**.

Exempel:

```
class Bil {
 final public function tuta() {
    echo "tuut";
  }

}
class Lastbil extends Bil {
  public function tuta() {
    echo "TUUT!";
  }
}
$fordon = new Lastbil();
$fordon->tuta();
```

Resultat: tuut

I exemplet ovan har superklassens (Bil) metod **`tuta()`** inte skrivits över av subklassens metod med samma namn pga. Nyckelordet **`final`**.

På liknande sätt så går det att skydda medlemsvariabler genom att skapa *klasskonstanter* som ej går att ändra på. Detta görs med reserverade ordet **`const`**.

Exempel:

```
class Frukt {
  const info = "Ätbar";

}
echo Frukt::info;
$banan = new Frukt();
echo $banan::info;
```

Resultat: ÄtbarÄtbar

I exemplet ovan skapas en *klasskonstant* som ej går att ändra på med det reserverade ordet **`const`**. Observera att konstanter börjar inte med **`$`** och man kommer åt dem med **`::`**
En annan viktig sak är att man behöver inte skapa ett objekt för att använda en klasskonstant. I exemplet används klasskonstanten dels direkt via klassen och dels via objektet **`$banan`**.

Som vi såg i förra exemplet så kan det vara användbart att komma åt saker i en klass utan att behöva skapa en instans av klassen (nytt objekt) utan att egenskapen eller metoden är en konstant. Detta går att åstadkomma med det reserverade ordet **`static`**.

Exempel:

```
class Bil {
public static function tuta() {
    echo "tuut";
  }
 public static $hjul=4;

}
Bil::tuta();
echo Bil::hjul;
```

Resultat: tuut 4

I exemplet ovan deklareras metoden **`tuta()`** och egenskapen **`$hjul`** som **`static`**. Detta gör dem till *klassgemensamma* metoder och egenskaper. De går att ändra på men är desamma oavsett hur många instanser som skapas av klassen. Statiska egenskaper och metoder går att använda utan att skapa instanser av klassen. Åtkomst sker med **`::`**